MASTERING YOUR HP-48

VOLUME 2 : PROGRAMMING AND APPLICATIONS

Jean-Michel FERRARD

D3I DIFFUSION

MASTERING YOUR HP-48

VOLUME 2 : PROGRAMMING AND APPLICATIONS

J.-M. FERRARD
Agrégé de Mathématiques
Professor of Advanced Mathematics
at Déodat de Séverac College, Toulouse

D3I
BP 49 — Tél. 61 73 81 82
31528 RAMONVILLE ST-AGNE CEDEX (FRANCE)

TABLE OF CONTENTS

NOTE

The information given in this book may be changed without prior notice.

D3I and HEWLETT-PACKARD cannot give any guarantee as regards the use of the formulae and examples shown in this book.

MASTERING YOUR HP48
Volume 2:
Programming and Applications

You are holding the second volume in the series "Mastering your HP48", which will help you to exploit
the power of your HP calculator to the full.

The **"Mastering your HP48"** series is in two volumes:

HP48: Programming and Exercises.
HP48: Programming and Applications.

Each volume covers different aspects:

* Volume 1 will help you to get to grips with programming your HP48. Separate chapters each deal with a specific problem and each chapter is followed by exercises for which answers and comments are given in the second half of the book.
* Volume 2 is a collection of programs designed to cater for the needs (mathematics in general) of all HP48 users.

The two volumes therefore complement each other to enable you to get the best out of your HP calculator.

The first volume ("Programming and Exercises") will have taught you - and, I hope enabled you to master - how to program the HP48.

In this second volume, I have compiled a vast library of "ready-to-run" programs to cater for the application you have in mind.

I hope that your reading of the first volume has made you feel at home enough with the calculator to be able to make any changes you may feel necessary to the programs that follow. I took a great deal of care in writing them and I therefore hope that you will get a lot of use out of them.

I myself have done my utmost to make the algorithms used short and quick.

However, nobody can claim to be perfect on that score, and the constant adjustments that I have made to my programs since I first wrote them make me think that you too will be able to improve them in one way or another.

But before going any further, let me say that I hope you will get as much pleasure as I have out of "taming" this demanding yet absorbing calculator.

THE STRUCTURE OF THIS BOOK

The book is divided into 15 chapters, each corresponding to a family of programmes to be installed in a specific directory.

1) **ARITHMETIC:** Gcd, Lcm, simplifying fractions, estimating rational expressions, decomposing into a product of prime factors, equations of the type ax + by = c in Z, etc.

2) **REAL AND COMPLEX NUMBERS:** infinite products, continued fractions. Iterative calculations. Trigonometric calculations. Rational approximations of the elements of a table. Calculations with the number j. Nth roots of a complex number, etc.

3) **POLYNOMIALS:** Operations on polynomials. Roots of a polynomial of degree $\leq$ 4. Bairstow's method. Polynomial arithmetic, etc.

4) **RATIONAL FRACTIONS:** Various operations on rational fractions, especially decomposition into partial fractions.

5) **MATRICES:** Basis changes. Powers of matrices. Calculating rank. Characteristic polynomial, eigenvectors and eigenvalues. Solving a system of equations symbolically. Pivot method. Linear relations and equations of vector sub-spaces.

6) **ANALYSIS TECHNIQUES:** Tangents, local extreme points, points of inflection. Least squares approximation. Non-linear systems. Fourier series. Differential equation, etc.

7) **FINITE SERIES:** Operations on finite series. Finite series of standard functions, composition of finite series, etc.

8) **AFFINE AND EUCLIDEAN GEOMETRY:** Equations, distances, angular distances.

9) **DIFFERENTIAL GEOMETRY:** Differential, divergence, Laplacean, gradient, curl. Length of a curve. Curvature. Area under a plane curve. Line integrals, etc.

10) **GRAPHS:** Plotting of parametric curves, curves with polar coordinates, family of curves. Envelope of a family of straight lines, etc.

11) **LARGE INTEGERS:** Operations on large integers; arithmetic.

12) **PROBABILITIES:** Probability distributions and distribution functions. Enumerations.

13) **SIMPLE STATISTICS:** Various means and characteristics of a simple statistic. Gini curve and coefficient. Histogram. Cumulative frequency polygon. 50% of cumulative mass, etc.

14) **STATISTICS IN TWO VARIABLES:** Means, variances of marginal statistics. Correlation, least squares straight lines, etc.

15) **DATABASES:** Creating a database and adding records. Reading, editing and sorting a database.

PROGRAM AND DIRECTORY NAMES

Each directory has its own specific name. This is also obviously true for the programs they contain.

Be careful if you decide to change any of these names. Certain programs make calls to others by calling their name. Likewise, certain programs may switch temporarily to another sub-directory. Again, the name of the sub-directory will figure in the caller program.

These are the names I have used for the directories:

1) Arithmetic: 'ARIT'
2) Real or complex numbers: 'R.C'
3) Polynomials: 'POLY'
4) Rational fractions: 'FRAC' (this directory must be installed in the 'POLY' directory)
5) Finite series: 'DL'
6) Matrix calculations: 'MATR'
7) Analysis techniques: 'ANAL'
8) Geometry: 'GEOM'
9) Differential geometry: 'GDIF'
10) Graphs: 'GRPH'
11) Large integers: 'LONG'
12) Probabilities: 'PROBA'
13) Simple statistics: 'STAT1'
14) Statistics in two variables: 'STAT2'
15) Databases: 'DATA'

WHICH PROGRAMS WILL YOU NEED?

If you haven't fitted your HP48 with extra memory, you will need to choose which programs to install in your calculator. The programs in this book will take up 42 Kbytes.

The programs you choose will depend on your specialist subject, your personal tastes and the memory taken up by the programs in the directories in question.

If you really have to choose, I think that the directories 'ARIT', 'R.C', 'POLY', 'FRAC', 'MATR' and 'FS' will prove useful to everybody.
The directories 'PROBA', STAT1' and 'STAT2' will be essential for students preparing for business school entrance examinations. Students preparing for engineering school entrance examinations will find 'ANLY' or 'GDIF' more useful.

For your information, the approximate amount of memory taken up by the programs in the various directories is given below (in bytes):

'ARIT'	:	1865	'R.C'	:	2120	'POLY'	:	5591
'FRAC'	:	951	'FS'	:	3626	'MATR'	:	4846
'ANLY'	:	3265	'GEOM'	:	1008	'GDIF'	:	2715
'GRPH'	:	1447	'LONG'	:	1941	'PROBA'	:	1791
'STAT1'	:	3314	'STAT2'	:	1509	'DATA'	:	1547

PROGRAM PRESENTATION:

Each chapter of this book sets out the programs of a specific directory, starting with a short introduction.

Each program is briefly presented. A flowchart shows the contents of the stack before the program is called up and after it has been run. Some extra comments may be given in the notes (does the program make calls to other programs?, etc.).
There was not enough space here to comment on the large number of programs presented (the book would have been twice the size). All programs are commented on at length in volume 1 of this series ("Programming and Exercises").

I have tried, as far as possible, to make the program listings easy to copy and easy to understand. For that reason:

* Words are clearly spaced.

* The program's structure is clearly shown using indents to distinguish between the start and end of loops, conditional structures, subroutines, etc. You do not of course need to stick to the same layout or use the same spacing when copying the listing.

* If a program uses a name for another program, a directory or a global variable that is required for the program to run correctly, and if that name is specific to this book (i.e. is not a known identifier used by the HP48), then it is underlined in the program listing in which it appears.

* All programs are illustrated by examples.

TIPS AND ADVICE:

* Create your directory (by going first to the 'HOME' directory) before going into it and installing the programs you require one at a time.

* Check that the programs run correctly using the examples I have given. A program 'A' may need to make a call to a program 'B', for example, in which case 'B' will have to be installed first if you want to run 'A'.

* If a program runs incorrectly, check it carefully against the reference listing.

* If -1 appears in a program, be careful not to confuse it with - 1. To enter -1 (with no space between), key in 1 then press +/-.

* Avoid cluttering your directories with variables or programs you no longer need. It is far more logical to create a specific directory for temporary applications and variables so that you can clean it up regularly (CLVAR Instruction).

* If a directory contains more than 6 entries, ensure that the most useful programs are at the top of the list and are grouped according to theme. This will enable you to work more efficiently. Remember to use the VARS and ORDER instructions to change the order in which programs appear in a given directory.

* Be careful when using the → sign, which appears in a large number of words in the language used by the HP48. It is better to write these words via the menus: for example, to write the words **→ARRY** and **ARRY→**, go through the **PRG OBJ** menu. The → sign is also used to create local variables, in which case it should be keyed in with the corresponding key on the HP48.

* I have used small letters to indicate local variables. Confusing a small letter with a capital letter would cause an error to occur. You can of course stick to capital letters only. If you do, you will need to check that there are no ambiguities between your local variables and certain identifiers used by the HP48. For example, it is quite possible to create a local variable and call it 'SIN', but this would make the HP's SINUS function temporarily inaccessible. Likewise, we can define a local variable called 'end' but we cannot use the name 'END'.

* Certain programs in this book use the instruction IFERR...THEN. I have assumed that the LASTARG function on your calculator is on (see user manual).

* For programs using trogonometric functions, I have assumed that your calculator is in radians mode (otherwise the results given by the programs will be different).

* For technical reasons, the character 0 (zero) is not barred in this edition. There is therefore a slight risk of mistaking the letter O for the figure 0, but the context should allow you to distinguish them clearly, and I have not called any variables O (the letter). An 0 on its own is therefore always the figure zero. But be careful all the same.

* For each program, I have given the size (in bytes) and the checksum obtained by using the BYTES instruction in the MEMORY menu for the program. The checksum lets you check that the program has been correctly copied.

Jean-Michel FERRARD

ARITHMETIC

The 'ARIT' directory contains programs written for integers and rational numbers.

You will find yourself using some of these programs on a regular basis, as they are extremely useful. To take an example, one of the programs allows us to evaluate an expression with rational numbers in it:

```
'1/25 + 27/11 - 3*51/4/9 + 13/7 - 9/5 + 22/17',
```

giving an almost immediate result as a simple fraction:

```
(-52909/130900).
```

You should install the following essential routines in the 'ARIT' directory:

'FCTR': resolving an integer into a product of prime factors.

'GCD': greatest common divisor of two integers.

'LCM': least common multiple of two integers.

'SIMP': simplifying a fraction.

'ABCXY': solving the equation ax+by=c in integers.

'CALC': evaluating rational expressions.

You may also install the following more specialized programs if you so wish:

'LPRM': list of prime divisors of an integer.

'MOEB': Moebius function.

'EULER': Euler index.

'CYCLO': calculating cyclotomic polynomials.

CALCULATING THE G.C.D.

'GCD' calculates the greatest common divisor of two integers a and b as shown in the calculation diagram below:

2:	a	'GCD'	2:	
1:	b	——>	1:	gcd(a,b)

'GCD': (checksum: # 31925d, size: 45 bytes)

```
«   WHILE   DUP
       REPEAT    SWAP   OVER   MOD   END
    DROP   ABS
»
```

<u>Example</u>:

2:	17660160	'GCD'	2:	
1:	9046890	——>	1:	19710

CALCULATING THE L.C.M.

'LCM' calculates the least common multiple of two integers a and b as shown in the calculation diagram below:

2:	a	'LCM'	2:	
1:	b	——>	1:	lcm(a,b)

N.B.: program 'LCM' calls program 'GCD':

'LCM': (checksum: # 43362d, size: 34 bytes)

```
«   DUP2   GCD   /   *   ABS   »
```

Exemple:

2:	45855	'LCM'	2:	
1:	50895	——>	1:	51862005

SIMPLIFYING A FRACTION

'SIMP' simplifies a fraction a/b (where a and b are both integers) as shown in the calculation diagram below (c/d represents the resulting simple fraction):

2:	a	'SIMP'	2:	c
1:	b	——————>	1:	d

N.B.: 'SIMP' calls program 'GCD'.

'SIMP': (checksum: # 10286d, size: 42.5 bytes)

```
«   DUP2   GCD   ROT   OVER   /   3   ROLLD   /   »
```

Example: (in less than one second)

2:	1659276	'SIMP'	2:	46091
1:	464508	——————>	1:	12903

RESOLVING THE EQUATION AX+BY=C (from Z)

'ABCXY' allows you to resolve the equation `AX+BY=C`, where A, B and C are given and X and Y are unknown, from the set Z of relative integers.
A sufficient necessary condition for solutions to be found - there are an infinite number - is that C be divisible by the GCD of A and B.
This gives us the following calculation diagram:

3:	A		3:	
2:	B	'ABCXY' ——>	2:	
1:	C		1:	Expression

where "Expression" describes the general solution to the problem. If no such solution exists, the error message "No solution" is displayed.

'ABCXY': (checksum: # 44788d, size: 306 bytes)

```
«  →  a  b  c
   «  0  1  b  3  →ARRY  1  0  a  3  →ARRY
      WHILE  DUP  3  GET
      REPEAT
         SWAP  DUP2  3  GET  OVER
         3  GET  /  FLOOR  *  -
      END
      DROP  c  OVER  3  GET  /  DUP
      IF  DUP  FLOOR  ==  THEN
         OVER  3  GET
         →  p
         «  *  OBJ→  DROP2  a  ROT  b  p  /
            'k'  *  +  *  b  ROT  a  p  /  'k'
            *  -  *  +  c  =
         »
         ELSE  DROP2  a  b  c  "No solution"  DOERR
         END
   »
»
```

Example: (in less than three seconds)

3:	23			
2:	17	'ABCXY' ——>	2:	
1:	1		1:	'23*(3+17*k)+ 17*(-4-23*k)=1'

Which means that the equation `23x+17y=1` is satisfied by the specific solution: x=3, y=-4 and that the general solution is:
`x = 3 + 17*k`
`y =-4 - 23*k`, where `k` is any relative integer.

FACTORIZING AN INTEGER

'FCTR' resolves an integer n into a product of prime factors as shown in the calculation diagram below:

```
1: [ n ]    'FCTR'
           -------->   1: [ resolution ]
```

where "resolution" is a list of complex numbers (p,k) each representing a prime integer p resolved from n and the corresponding exponent k.

'FCTR': (checksum: # 50846d, size: 231 bytes)

```
« ABS {} SWAP 1
  → n k
  « WHILE n 1 >
    REPEAT
      IF k SQ n <
        THEN k DUP 2 > + 1 +
        ELSE n
      END
      'k' STO
      IF n k MOD NOT THEN
        k 0 n
        WHILE DUP k MOD NOT
        REPEAT
          k / SWAP 1 + SWAP
        END
        'n' STO R→C +
      END
    END
  »
»
```

Example: (in less than 2 seconds)

```
1: [ 147840 ]  'FCTR'
               ----->  1: [ { (2,7) (3,1) (5,1) (7,1) (11,1)} ]
```

Since 147840 = 2^7*3*5*7*11.

Example: (in 19 seconds)

```
1: [ 2840121 ]  'FCTR'
                ------>  1: [ { (3,2) (315569,1) } ]
```

Since 2840121 = 3^2*315569.

EVALUATING RATIONAL EXPRESSIONS

'CALC' evaluates an algebraic expression consisting of sums, differences, products and integer quotients, giving the solution as a simple fraction a/b. Program 'CALC' also calls program 'SIMP'.

Calculation diagram:

```
2: |              |    'CALC'     2: |   a   |
                     ----------->
1: | 'expression' |               1: |   b   |
```

'CALC': (checksum: # 62365d, size: 300 bytes)

```
«  DUP   EVAL   SWAP   →STR   1
   →  v  c  d
   «  WHILE  c  DUP  "/"  POS  DUP
      REPEAT
         1  +  OVER  SIZE  SUB  DUP  NUM
         IF  10  ==  THEN  2  OVER  SIZE  SUB  END
         'c'  STO  0
         WHILE  "0123456789"  c  1  1  SUB  POS  DUP
         REPEAT
            1  -  SWAP  10  *  +  c
            2  OVER  SIZE  SUB  'c'  STO
         END
         DROP  'd'  STO*
      END  DROP2
      v  d  *  .5  +  FLOOR  d  SIMP
   »
»
```

Example: (in 2 seconds)

```
2: |                            |    'CALC'     2: |  -1171  |
                                   ----------->
1: | '1/6-15/27+2/11+4/15/7'    |               1: |  6930   |
```

Caution!
For the program to run correctly, each / sign in the expression to be evaluated must be followed by a figure (and not an open bracket, for example).
Thus, `'5/(11/3)'` must be written `'5/11*3'` or `'15/11'`. Likewise, denominators must not be raised to a power.
Thus `'3/2^5'` must be written `'3/32'`.

Program 'CALC' calculates the product of all denominators in the expression. A round-off error will therefore occur if the product is greater than 1E12.

PRIME DIVISORS OF AN INTEGER

'LPRM' gives the list of all positive prime divisors of an integer n, as shown in the calculation diagram below:

```
      |              |       'LPRM'            |               |
 1:   |      m       |   -------------->   1:  |    {list}     |
```

N.B.: Program 'LPRM' calls Program 'FCTR'.

'LPRM': (checksum: # 36961d, size: 61 bytes)

```
«     FCTR
      IF     DUP      SIZE     THEN
             OBJ→     →ARRY    RE
             OBJ→     1        GET     →LIST
      END
»
```

Example: (one second)

```
      |              |       'LPRM'            |               |
 1:   |    518400    |   -------------->   1:  | { 2   3   5 } |
```

(in fact, `518400 = (2^8)*(3^4)*(5^2) ).`

N.B.: Program 'LPRM' is called by 'CYCLO' and 'EULER'.

CYCLOTOMIC POLYNOMIALS

The cyclotomic polynomial Φ_n of order n is the unitary polynomial whose zeros are the nth primitive roots of unity, i.e.

$$W_k=\cos(2k\pi/n) + i \sin(2k\pi/n),$$

where k includes the integers between 1 and n that are prime to n. The degree of the polynomial Φ_n is $\varphi(n)$ where φ is the Euler index.

This gives us the calculation diagram below:

$$1: \boxed{\quad n \quad} \xrightarrow{\text{'CYCLO'}} 1: \boxed{\quad \Phi_n \quad}$$

The result obtained is a vector representing the components of the polynomial, in decreasing order of the powers of the unknown X.

Example: (in 4 seconds)

$$1: \boxed{\quad 10 \quad} \xrightarrow{\text{'CYCLO'}} 1: \boxed{[\ 1\ \ -1\ \ 1\ \ -1\ \ 1\ ]}$$

since $\Phi_{10}(X)=$ X^4 -X^3 +X^2 -X +1.

Example: the polynomial Φ_{30} of degree 8 is obtained in 13 seconds. The polynomial Φ_{105} of degree 48 is obtained in 1 min. 5 s.

The program 'CYCLO' uses the formula:

$$\Phi_n = \prod_{d|n} (x^d - 1)^{\mu\left(\frac{n}{d}\right)}$$

where μ is the Moebius function and where the product includes all positive divisors of n.

N.B.: Program 'CYCLO' calls program 'LPRM'.

TEXT OF PROGRAM 'CYCLO'

'CYCLO': (checksum: # 20046d, size: 643 bytes)

```
«   DUP    {}    0    0
    →    n    m    dp    k    pr
    «    IF    n    1    ==    THEN    [ 1 -1 ]
         ELSE
              «    0
                   1    dp    SIZE    FOR    i
                        k    dp    i    GET    MOD    NOT    +
                   NEXT
                   2    MOD
              »
              «    →    w
                   «    1    OVER    SIZE    1    GET    n    k    /    –
                        IF    w    -1    ==    THEN    SWAP    END
                        FOR    i
                             n    k    /    i    +    DUP2    GET
                             3    PICK    i    GET    w    *    +    PUT
                        w    STEP
                   »
              »
              «    OVER    SIZE    1    GET    n    k    /
                   ROT    *    –    1    →LIST    RDM
              »
              →    ip    dm    rd
              «    m    LPRM    DUP    'dp'    STO
                   OBJ→    1    1    ROT    START    *    NEXT
                   'pr'    STO
                   [ 1 ]
                   pr    1    FOR    k
                        IF    pr    k    MOD    NOT    THEN
                             IF    ip    EVAL    NOT    THEN
                                  -1    rd    EVAL    -1    dm    EVAL
                             END
                        END
                   -1    STEP
                   1    pr    FOR    k
                        IF    pr    k    MOD    NOT    THEN
                             IF    ip    EVAL    THEN
                                  1    dm    EVAL    1    rd    EVAL
                             END
                        END
                   NEXT
              »
         END
    »
»
```

EULER INDEX

The Euler index $\varphi(n)$ of a natural integer n is equal to the number of natural integers p between 1 and n and prime to n.

If we write the resolving of n into prime factors as:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots . \; p_k^{\alpha_k}, \quad \text{then } \varphi(n) \text{ is equal to:}$$

$$n \left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \ldots . \left(1 - \frac{1}{p_k}\right).$$

In particular, if n is prime, then $\varphi(n) = n-1$.

'EULER' calculates the Euler index of the integer n as shown in the calculation diagram below:

```
1: [ n ]    --'EULER'-->    1: [ φ(n) ]
```

N.B.: Program 'EULER' calls program 'LPRM'.

'EULER': (checksum: # 61654d, size: 93 bytes)

```
«   IF   DUP   ABS   1   >   THEN
    →    n
    «    n   LPRM   OBJ→   n   1   ROT   START
             DUP    ROT    /   -
         NEXT
    »
    END
»
```

Ex/ample: (1 second)

```
1: [ 420 ]    --'EULER'-->    1: [ 96 ]
```

MOEBIUS FUNCTION

The Moebius function μ is defined for the set of non-zero natural integers. If n is a non-zero natural integer,

$\mu(n) = 0$ if n is divisible by the square of a prime integer.

If n is not divisible by the square of a prime integer, then μ is equal to 1 or -1 depending on whether the number of prime divisors of n is even or odd.

'MOEB' calculates μ(n) as shown in the calculation diagram below:

```
1: [ n ]   --'MOEB'-->   1: [ µ(n) ]
```

N.B.: 'MOEB' calls program 'FCTR'.

'MOEB': (checksum: # 27678d, size: 101.5 bytes)

```
«   FCTR   OBJ→
    →   n
    «   IF   n
           THEN
                n   →ARRY   IM   RNRM   2   <   1
                n   2   MOD   2   *   -   *
           ELSE   1
        END
    »
»
```

For example, we find:

$\mu(1)=1$, $\mu(2)=-1$, $\mu(4)=0$, $\mu(6)=1$

$\mu(30)=-1$ (in 1 second) $\mu(210)=1$ (in 1 second).

REAL NUMBERS AND COMPLEX NUMBERS

The directory devoted to real or complex numbers contains programs that you can use regularly and which are related only by the fact that they do not fit easily into any of the more specialized directories.

The 'R.C' directory contains the following programs:

- **'PROD'**: Partial products of an infinite product.
- **'CNFR'**: calculation of continued fractions and reduced fractions of a given real number.
- **'A→Q'**: approximation, in the same manner as the instruction →Q, of the elements of an array of real or complex coefficients.
- **'I→J'**: writing of a complex number in the form a+bj.
- **'ITER'**: allows you to do recursive calculations (with any size step required).
- **'NRCN'**: nth roots of a real or complex number.
- **'TRIG'**: linearization of the powers of cos(t) and sin(t), and the inverse operation to linearization.
- **'INTZ'**: integration along a line in the complex plane.

CALCULATING PARTIAL PRODUCTS

'PROD' calculates the product $\prod_{k=m}^{k=n} u_k$ where u_k is a real number depending on k.
m and n are the two integers representing the bounds of the index of the product k.

The product is calculated for the value k=m (the value at level 1 in the stack) and arrives at the value k=n (at level 2). (It may be that m>n, in which case the values of k are given in decreasing order. The advantage for the user is that there is no need to worry about the order in which m and n are given. Another advantage is that we can thus control the order of the product (to get the most accurate results out of the calculator, it is best to multiply values closest to 1 first.)).

This gives us the following functional diagram:

3:	u(N)				
2:	n	'PROD' ——>			
1:	m			1:	$\prod_{k=m}^{k=n} u$

(where u(N) is an algebraic expression or a program that can be used to evaluate the term u_N. A capital "N" must be used here).

'PROD': (Checksum: # 32428d, Size: 108 bytes)

```
«   DUP2    >    2    *    1    -
    "« → N"   5    ROLL    →STR    +    OBJ→
    →    s    u
    «    1    SWAP    ROT    FOR    N
              N    u    →NUM    *
         s    STEP
    »
»
```

Example: (in 6 seconds)

3:	'EXP(1/N)'		3:	
2:	200	'PROD' ——>	2:	
1:	100		1:	2.0150689848

(if in the case above we reverse the order of 100 and 200, i.e. if we calculate the product of k=200 down to k=100, the result obtained is 2.01506898486. This result is without doubt the most accurate).

CALCULATING CONTINUED FRACTIONS

Let x be a real number and a_1 its integer part. If x is not an integer, we can write:
$x = a_1 + 1/x_1$ where $x_1 > 1$

Let a_2 be the integer part of x_1. If x_1 is not an integer, we can write:
$x_1 = a_2 + 1/x_2$ where $x^2 > 1$ and therefore:

$$x = a_1 + \frac{1}{a_2 + 1/x_2}$$

If we pursue this operation, we obtain a series of integers a_1, a_2, ..., a_n and a series of real numbers x_1, x_2, ..., such that:

$$x = a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots \; a_n + \cfrac{1}{x_n}}}}$$

The series is finite if x is a rational number (one of the values of x_n is an integer), otherwise it is infinite (no value of x_n is an integer).
The quantity shown above is called a continued fraction of x.
The values of a_n are called the partial quotients of the continued fraction.
The continued fraction:

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots \; a_{n-1} + \cfrac{1}{a_n}}}}$$

is written $[a_1/a_2/..../a_n]$ and called the nth reduced fraction of x. It is a rational number r_n expressed as $r_n = p_n / q_n$ where p and q are two relatively prime integers.
The reduced fractions r_n of x are useful approximations of x.
We can show that (if x is not a rational number):

- The series of reduced fractions of x converges to x.
- The series p_n and q_n converge to infinity.
- x is always between two consecutive reduced fractions.
- the difference between x and its nth reduced fraction r_n is such that:

$$| x - r_n | < \frac{1}{q_{n-1} \; q_n}$$

CONTINUED FRACTIONS (cont.)

The functional diagram for 'CNFR' is as follows:

```
2:   x              'CNFR'      2:   list2
                 ---------->
1:   n                          1:   list1
```

where:

* x is the real number for which we want to obtain reduced fractions.
* n is the number of times we want to repeat the calculation of those reduced fractions.
* list1 is the list of partial quotients a_k of x (up to n).
* list2 is t he list of reduced fractions $p_k\ /\ q_k$ (in the form (p_k,q_k)) up to $p_n\ /\ q_n$.

'CNFR': (Checksum: # 56779d, Size: 206.5 bytes)

```
«  {}   (1,0)   (0,1)
   →    x    n    k    r    s
   «    1    n    START
             x    FLOOR   'k'   OVER   STO+
             x    OVER    -     INV    'x'   STO   r   *
             s    +    r  's'   STO    'r'   STO   r
        NEXT
        n    →LIST   k
   »
»
```

Example:

We want to find 6 reduced fractions of π. In one second, we obtain:

```
                                  { (3,1) (22,7) (333,106) (355,113)
2:  3.14159265359     'CNFR'         (103993,33102) (104348,33215)  }
                     ------->
1:        6                   1:      { 3 7 15 1 292 1 }
```

Meaning that the successive reduced fractions of π are:

```
[ 3 ] = 3
[ 3 / 7 ]= 3 + 1/7 = 22/7 ≈ 3.14285714286.
                        1
[ 3 / 7 / 15] = 3 + ---------  = 333 / 106 ≈ 3.14150943396.
                     1 + 1/15
[ 3 / 7 / 15 / 1 ] = 355 / 113 ≈ 3.14159292035.
[ 3 / 7 / 15 / 1 / 292 ] = 103993 / 33102 ≈ 3.14159265301.
[ 3 / 7 / 15 / 1 / 292 / 1] = 104348 / 33215 ≈ 3.14159265392.
```

RATIONAL APPROXIMATION OF A REAL OR COMPLEX ARRAY

'A→Q' calculates an approximation, using rational numbers, of the various elements of a vector or of a matrix of real or complex coefficients, using the instruction →Q.

The accuracy of the approximation depends on the display mode used:

* If the **n FIX** mode is on, the approximation is correct to n decimal points.
* If the **STD** mode is on, the approximation is correct to 12 significant figures.

The functional diagram is as follows:

```
        ┌─────────┐       'A  Q'            ┌─────────┐
     1: │    A    │     ──────────>      1: │  list   │
        └─────────┘                         └─────────┘
```

where "list" contains the elements obtained from the various approximations. For a matrix A, "list" is made up of the sub-lists of each row of the matrix A.

If you switch to **n FIX** mode, the integer n must be large enough for the rational approximation not to be too rough.
It must, however, take into account any round-off errors that may affect the numbers we want to approach using rational numbers.
As a rule, values between n=8 and n=10 give correct results.

'A→Q': (Checksum: # 25770d, Size: 183.5 bytes)

```
«  OBJ→   OBJ→   IF   1   ==   THEN   1   SWAP   END
   →  lig   col
   «  1   lig   FOR   i
         1   col   START
            →Q   col   ROLLD
         NEXT
         col   →LIST
         col   lig   i   -   *   i   +   ROLLD
      NEXT
      IF   lig   1   >   THEN   lig   →LIST   END
   »
»
```

PROGRAM 'A→Q': PRACTICAL EXAMPLES

Example 1:

We want to resolve the system:

```
| 2x +  y -  z = 1
|  x - 3y + 5z = -3
| 3x + 5y -  z = 0
```

into rational numbers.

We put `[1,-3,0]` at level `2` and

```
[[ 2  1 -1 ]
 [ 1 -3  5 ]
 [ 3  5 -1 ]]
```

at level `1`.

We do the operation / and find the solution, in real numbers, at level 1.
We then switch to **10 FIX** mode and call '→Q'.
This gives us, in two seconds, at level 1:

```
{ '5/21' '-(13/42)' '-(5/6)' }
```

Which proves that the exact solution to the system above is:

```
x = 5/21, y = -13/42, z = -5/6.
```

Example 2:

We want to find the inverse of the matrix `A =`

```
[[  1  3 -4 ]
 [  8 -2  3 ]
 [ -5  1  7 ]
```

We put the matrix at level 1, then do the operation '1/x'.
We then switch to **9 FIX** mode and call the program 'A→Q'.

In about 3 seconds, we obtain the following list:

```
{{ '17/222' '25/222' '-1/222' }
 { '71/222' '13/222' '35/222' }
 { '1/111'  '8/111'  '13/111' } }.
```

The inverse of A is therefore `1/222*`

```
[[ 17  25  -1 ]
 [ 71  13  35 ]
 [  2  16  26 ]]
```

Example 3:

This example uses certain programs in the 'FS' directory.
We want to find the exact 5th order finite series, converging to zero of `Tan(Ln(1+X))`.
We first go to the 'FS' directory.
We put 5 in the stack, then call the programs 'LG' and 'TG' one after the other. We thus find the required series in real-number form.
We then go back to the 'R.C' directory and switch to **10 FIX** mode.

We then call 'A→Q' and obtain the list:

```
{  0  1  '-1/2'  '2/3'  '-3/4'  '11/12' }
```

Which proves that the finite series we want to find is written:

```
Tan(Ln(1+X)) = X -1/2*X^2 +2/3*X^3 -3/4*X^4 +11/12*X^5 +o(X^5)
```

WRITING A COMPLEX NUMBER IN THE FORM a+bj.

'I→J' is used to write a complex number z in the form a+bj, where a and b are two real numbers and j is the complex number with a modulus of 1 and an argument of $2*\pi/3$.

The complex number z may be written in the form of an algebraic expression. In fact, 'I→J' is especially designed to perform rapid calculations using the number j. It is therefore highly recommended to create a variable j in the 'R.C' directory for this purpose.

To create this variable, follow the sequence below:

```
1   ENTER   NEG   3   √   R→C   2   /   'j'   STO
```

The functional diagram for 'I→J' is as follows:

```
                 'I→J'
1:  [    z    ]  ------>  1:  [ expression ]
```

where "expression" is in the form a+bj, and a and b are two real numbers.

' I→J': (Checksum: # 8929d, Size: 62 bytes)

```
«   →NUM   C→R   3   √   /   DUP   2   *
    'j'    *     3   ROLLD  +  SWAP  +
»
```

<u>Examples</u>: (in one second)

Both the examples below assume that a variable j has been created.

```
                            'I→J'
1:  '(3+4*j)/(11-2*j)'  ------>  1:  '.210884353742+.340136054422*j'
```

Switching to 10 FIX mode and running →Q, we obtain:

`'31/147+50/147*j'`

```
                     'I→J'
1:  '(2+3*j)^5'  ------>  1:  '149.000000002+87.0000000012*j'
```

I.e.: `(2+3*j)^5 = 149+87*j.`

ITERATIVE CALCULATIONS

'ITER' allows you to perform iterative calculations with as many steps as required.
We first take a recurrence relation:

$$U_n = F(U_{n-p}, U_{n-p+1}, \ldots, U_{n-1})$$

allowing us to calculate the "objects" U_n, the series U being initialized by giving the p initial terms $U_0, U_1, \ldots, U_{p-1}$.
This will enable us to calculate the following terms. This general type of problem can be solved using the program 'ITER'.
For 'ITER' to run correctly, a variable called "LIST" must be entered in the directory in the format shown below:

{ N U_{N-p}, U_{N-p+1}, ..., U_{N-1} F(U_{N-p}, U_{N-p+1}, ..., U_{N-1})

Once 'ITER' is called, it creates the variables `N, U0, U1 .... U(p-2), U(p-1)` in the directory.
'ITER' then halts and the value of U_N is given.
If we press CONT, the next value U_{N+1} is calculated and the program halts again. We can then repeat the operation as many times as we want to obtain the successive values of the series U.

To interrupt the program, it is best to empty the stack before pressing CONT. 'ITER' is thus able to understand that the user has finished calculating and empties the directory of all the variables that had been created.

'ITER': (Checksum: # 60091d, Size: 358 bytes)

```
«   LIST    SIZE
    «    "'U"    SWAP    →STR    +    OBJ→    »
    →    d    var
    «    LIST    DUP    1    GET    'N'    STO    2
         0    d    2    -    FOR    i
              GETI    i    var    EVAL    STO
         NEXT
         DROP2
         DO
              d    2    -    var    EVAL    EVAL    EVAL    HALT
              IF    DEPTH    NOT    THEN
                   'N'    PURGE
                   0    d    2    -    FOR    i
                        i    var    EVAL    PURGE
                   NEXT
                   0    DOERR
                ELSE
                   'N'    1    STO+
                   1    d    2    -    FOR    i
                        i    var    EVAL    RCL    i    1    -    var    EVAL    STO
                   NEXT
                   d    3    -    var    EVAL    STO
              END
         UNTIL    0    END
    »
»
```

PROGRAM 'ITER': PRACTICAL EXAMPLES

Example 1:

We want to calculate the successive terms of the series whose general term is u and is defined by the recurrence relation:

$$u_n = \frac{u_{n-1}}{2} + \frac{2}{u_{n-1}}$$

and the initial term u = 1.
We put { 1 1 'U0/2+2/U0' } in 'LIST' and call 'ITER'.
After one second (while the variables are created) the program halts and gives the value of u_1, i.e. 2.5; we press CONT to obtain the value of u_2, i.e. 2.05; the following values are:

$u_3 = 2.0006097561$, $u_4 = 2.00000009292$, then $u_5 \approx 2.$

The stack must be purged before pressing CONT if we want to interrupt the program by purging the directory of all intermediate variables.

Example 2:

We want to calculate the successive terms of the matrix series M defined by the recurrence relation:

$$M_n = M_{n-1} - n*^t (M_{n-2}).$$

and the initial terms

```
     [[ 1 0 0 ]               [[ 0 1 0 ]
M0 =  [ 0 1 0 ]          M1 =  [ 1 0 1 ]
      [ 0 0 1 ]]               [ 0 1 0 ]]
```

We put { 2 [[1 0 0 [0 1 0 [0 0 1]] [[0 1 0 [1 0 1 [0 1 0]] « U1 U0 TRN N * - » } in the variable 'LIST' then call 'ITER'.

```
                                          [[ -2 1 0 ]
The value of M2 is rapidly calculated, i.e. M2 = [ 1 -2 1 ]
                                           [ 0  1 -2]]
```

The following matrices in the series M are given by pressing CONT to obtain each subsequent matrix.

Example 3:

We want to calculate the successive terms of the series of polynomials defined by the recurrence relation:

$P_n (x) = nP_{n-1} (X) - xP_{n-2} (x) + x^2 P'_{n-3} (x).$

and the initial terms:

$P_0 (x) = 0$, $P_1 (x) = x$ and $P_2 (x) = x^2.$

We put, in the variable 'LIST':

```
{ 3 [0] [1 0] [1 0 0] « N  U2  U1  U0  POLY  →  n  u2  u1  u0  « u2
n  *  u1  [-1 0]  PRODP  ADDP  u0  DERIV [1 0 0]  PRODP ADDP R.C »}
```

(note the switch to the 'POLY' directory, where the local variables u0, u1, u2, u3, and n are created, before returning to the 'R.C' directory).

Nth ROOTS OF A COMPLEX NUMBER

'NRCN' calculates the nth roots x_0, x_1, . . . , x_{n-1} of a complex number z as shown in the functional diagram below:

			n:	x_0
			. . .	. . .
4:			4:	x_{n-4}
3:			3:	x_{n-3}
2:	z	'NRCN'	2:	x_{n-2}
1:	n	———>	1:	x_{n-1}

The formula used is:

if $z = R \exp(\ i\ \theta\ )$, then $x_k = \sqrt[n]{R} \exp(\ i(\ \theta/n + 2k\pi/n\ )\ )$.

'NRCN': (Checksum: # 5849d, Size: 81 bytes)

```
«   -1   OVER   INV   2   *   ^
    →   n   w
    «   n   INV   ^
        2   n   START
            DUP   w   *
        NEXT
    »
»
```

Example:

4:			4:	(2,-1)
3:			3:	(1, 2)
2:	(-7,-24)	'NRCN'	2:	(-2, 1)
1:	4	———>	1:	(-1,-2)

LINEARIZATION AND INVERSE OF LINEARIZATION

'TRIG' lets you express `cos(t)^n` and `sin(t)^n` in terms of quantities of the type `cos(pt)` and `sin(pt)`: this is the principle of linearization.

Conversely, 'TRIG' lets you express `cos(nt)` and `sin(nt)` in terms of the powers of `cos(t)` and `sin(t)`.

Note: program 'TRIG' calls programs 'TCHEB' and 'V→P', which must be in the 'POLY' directory.

When you call 'TRIG', the following menu appears:

COS^	SIN^	COS()	SIN()		EXIT

and the program is halted.

Press "EXIT" if you wish to quit the program.

You first enter an integer n at level 1 then press one of the first 4 keys to perform one of the following operations:
* linearization of `cos(t)^n`.
* linearization of `sin(t)^n`.
* express `cos(nt)`.
* express `sin(nt)`.

Example: if we put 3 at level 1 in the stack, we obtain at level 1:

`'COS(T)^3=(COS(3*T)+3*COS(T))/4'` by pressing **COS^**

`'SIN(T)^3=(-SIN(3*T)+3*SIN(T))/4'` by pressing **SIN^**

`'COS(3*T)=4*COS(T)^3-3*COS(T)'` by pressing **COS( )**.

`'SIN(3*T)=-(4*SIN(T)^3)+3*SIN(T)'` by pressing **SIN( )**.

Example: with n=15, results are computed within 3 to 9 seconds. We find, for example:

```
'SIN(15*T)=-(16384*SIN(T)^15)+61440*SIN(T)^13-92160*SIN(T)^11
+70400*SIN(T)^9-28800*SIN(T)^7+6048*SIN(T)^5
-560*SIN(T)^3+15*SIN(T)'
```

TEXT OF PROGRAM 'TRIG'

'TRIG': (Checksum: # 49045d, Size: 984.5 bytes)

```
«  0  0  →  n  k
   «  «  n  k  COMB  n  k  2  *  -  »
      «  IF  DUP  THEN  'T'  *  COS  *
         ELSE  DROP  2  /  END  »
      «  2  n  1  -  ^  /  »
      «  IF  2  MOD  THEN  NEG  END  »
      →  p1  p2  p3  p4
      «  {  {  "COS^"
               «  →  n  «  'COS(T)'  n  ^  0
                     0  n  2  /  FLOOR  FOR  k
                        p1  EVAL  p2  EVAL  +
                     NEXT  p3  EVAL  =
               »  »  }
            {  "SIN^"
               «  →  n  «  'SIN(T)'  n  ^  0
                     0  n  2  /  FLOOR
                     IF  n  2  MOD  THEN  FOR  k
                        p1  EVAL  'T'  *  SIN  *  k  n
                        1  -  2  /  +  p4  EVAL  +
                     NEXT
                     ELSE  FOR  k
                        p1  EVAL  p2  EVAL  k  n
                        2  /  +  p4  EVAL  +
                     NEXT
                     END  p3  EVAL  =
               »  »  }
            {  "COS()"
               «  →  n  «  n  'T'  *  COS  n  POLY  1
                     TCHEB  'COS(T)'  V→P  R.C  =
            »  »  }
         {  "SIN()"
            «  →  n  «  n  'T'  *  SIN
                  IF  n  2  MOD  THEN
                     n  1  POLY  TCHEB  n  1  -  2  /
                     p4  EVAL  'SIN(T)'  V→P  R.C
                  ELSE
                     IF  n  DUP  THEN
                        1  -  2  POLY  TCHEB  n  2  /
                        1  +  p4  EVAL  'SIN(T)'  V→P
                        R.C  'COS(T)'  *
                     END
                  END  =
            »  »  }
         {}
         {  "EXIT"  «  CONT  »  }
      }
      TMENU  HALT  2  MENU
»  »  »
```

INTEGRATION ALONG A LINE IN THE COMPLEX PLANE

'INTZ' calculates $\int_{\Gamma} f(z)dz$, where f is a function of the complex variable z and Γ is an arc with the coordinates $x=x(t),\ y=y(t),\ a \le t \le b$.

This gives the functional diagram below:

2:		'INTZ'	2:	
1:	list	————————>	1:	integral

where "list" is a list containing the data required for the calculation to be performed and "integral" is a value approaching the result (a complex number).

The accuracy of the calculation depends on the display mode used:
In **STD** mode, the calculation is performed as accurately as is possible (which will probably take time).

In **n FIX** mode, it is will be correct to n decimal places.

The format of "list" is `{ F Z A B }`, where:

F: expresses the function f, with respect to the variable 'Z'.

Z: is the expression with respect to the variable 'T' in symbolic form, giving the parameters of the arc Γ.
Z will be evaluated as a complex number representing the coordinates of the point (x(t), y(t)) on the arc.

A,B: are the bounds of the parameter T.

'INTZ': (Checksum: # 44859d, Size: 133 bytes)

```
«  EVAL   ROT   DUP   'Z'   STO
   'T'   DUP   PURGE   ∂   4   ROLL   'T'   SHOW
   *   3   DUPN   RE   'T'   ∫   →NUM   4   ROLLD
   IM   'T'   ∫   →NUM   R→C   'Z'   PURGE
»
```

PRACTICAL EXAMPLES WITH PROGRAM 'INTZ'

Example 1:

Calculate $\int_{\Gamma} (1/Z)\ dZ$, where Γ is the directed arc:

`X(T)= EXP(T), Y(T)=T*LN(T), 1≤T≤2.`

We switch, for example, to 6 FIX mode.
We then enter `{ '1/Z' 'EXP(T)+i*T*LN(T)' 1 2 }` at level 1 of the stack and call 'INTZ'.

After 25 seconds we obtain:

```
1: ( 1.017297 , 0.185459 )
```

Example 2:

We know that $\int_{\Gamma} f(Z)\ dZ$ is purely a function of the bounds and direction of: of the path of Γ, provided that the curve in Γ is continuous and within a domain of the complex field where f is holomorphic.

This can be seen if we integrate `f(Z)=EXP(Z)` between the point `A(1,0)` and the point `B(0,1)`:
1) along the segment joining A to B.
2) along the arc of the unit circle joining A and B.

We switch, for example, to 5 FIX mode.
We then put in at level 1 of the stack:
In the first case: `{ 'EXP(Z)' '1-T+i*T' 0 1 }`,
and in the second case: `{ 'EXP(Z)' 'EXP(i*T)' 0 'π/2' }`.
and we call up 'INTZ'.

In the first case, we obtain the following in 11 seconds:

```
1: (-2.17798,0.84147)
```

In the second case, we obtain the following in 27 seconds:

```
1: (-2.17798,0.84147)
```

The exact result is:

`exp(i)- exp(1) ≈ ( -2.17797952259, .841470984808 ).`

POLYNOMIALS

The 'POLY' directory contains programs used for polynomials with one unknown (with real or complex coefficients).

The programs listed below are some of the most useful and will relieve the user of time-consuming tasks (where calculation errors frequently occur).

Polynomials will be represented here by the vector of their components in decreasing order of powers of the unknown.

The polynomial `P = 3X^7-2X^6+X^3-2X` is thus represented by the vector:

`[ 3 -2 0 0 1 0 -2 0 ].`

Polynomials that are arguments in a program must be written in the above form, which is also the form in which they will be obtained if they are the results of a program.

Exceptions:

- The polynomials that are arguments in program 'DIVIP' (division in increasing order of powers) must be represented by the vector of their components in increasing order of powers of the unknown),
- In program 'REV' (which switches from increasing to decreasing powers).
- In program V→P (transformation of a polynomial written in 'vector' form into a more algebraic notation).

The 'POLY' directory is extremely complete. It includes:

'ADDP'	:	Addition of two polynomials.
'PRODP'	:	Product of two polynomials.
'POWP'	:	Raising of a polynomial to an integer power.
'DIV'	:	Euclidean division of two polynomials.
'DIVIP'	:	Division, in increasing order of powers, of two polynomials.
'COMP'	:	Composition of two polynomials.
'TRNS'	:	Translation of a polynomial.

'DEG2' : Real or complex roots of a 2nd degree polynomial.

'DEG3' : Real or complex roots of a 3rd degree polynomial.

'DEG4' : Real or complex roots of a 4th degree polynomial.

'BRST' : Real or complex roots of a polynomial of any degree (Bairstow's iterative method).

'VALP' : Value of a polynomial at a point.

'REV' : Reversing the order of the components of a vector

'DERIV' : Derivative of a polynomial.

'PRIM' : Primitive (cancelling to zero) of a polynomial.

'INTP' : Integral of a polynomial from a to b.

'GCDP' : GCD of two polynomials.

'LCMP' : LCM of two polynomials.

'TCHEB' : Calculates Tchebyshev polynomials of the first or second kind.

'V→P' : Transforms a polynomial (written in vector form) into conventional algebraic notation.

'PPCS' : Primitive, in symbolic form, of an expression of the type: `P(x)Cos(ax)+Q(x)Sin(ax)`, where P and Q are both polynomials.

'PPEX' : Primitive, in symbolic form, of an expression of the type: `P(x)Exp(ax)`, where P is a polynomial.

'PMAT' : Calculates matrix polynomials.

'EXPPP' : Expansion of a product of polynomial powers.

'ABCUV' : Finding a solution to the equation `AU+BV=C`, where A, B and C are three given polynomials and U and V are two unknown polynomials.

'ELML', 'ELMR' : are 2 routines used for operations on polynomials where the aim is to compensate for certain round-off errors.

ADDITION OF TWO POLYNOMIALS

'ADDP' calculates the sum of two polynomials P and Q as shown in the functional diagram below:

```
    |            |                        |            |
 2: |     P      |      'ADDP'        2:  |            |
    |            |   ------------->       |            |
 1: |     Q      |                    1:  |    P+Q     |
```

N.B: 'ADDP' is useful when the vectors representing P and Q are of different length, or if we do not know these lengths a priori (otherwise it is better to use +).

N.B: 'ADDP' calls program 'ELML', which deletes any zeros to the left of the resultant vector when adding two polynomials of the same degree (see example 2).

'ADDP': (Checksum: # 35585d, Size: 151 bytes)

```
«  IFERR   +   ELML   THEN
        OVER   SIZE   1   GET   OVER   SIZE   1   GET   -
        IF   DUP   0   <   THEN   ABS   ROT   SWAP   END
        →   a   d
        «   1   d   START   0   NEXT
            a   OBJ→   1   GET   d   +   →ARRY   +
        »
   END
»
```

Example 1: (less than one second)

```
 2: | [ 1 2 3 4 5 ] |      'ADDP'        2: |                    |
                       ------------->
 1: |  [ 10  100 ]  |                    1: | [ 1 2 3 14 105 ]   |
```

(since, if `P = X^4+2X^3+3X^2+4X+5` and `Q = 10X+100`, then:
`P+Q = X^4+2X^3+3X^2+14X+105`)

Example 2: (less than one second)

```
 2: | [ 1 2 3 4 5 ] |      'ADDP'        2: |             |
                       ------------->
 1: | [-1 -2 1 1 1 ]|                    1: |  [ 4 5 6 ]  |
```

Here we could quite simply have used the + operation, but the resultant would have been:

`[ 0 0 4 5 6 ].`

PRODUCT OF TWO POLYNOMIALS

'PRODP' calculates the product of two polynomials A and B as shown in the functional diagram below:

```
        |          |                    |          |
        |----------|                    |----------|
     2: |    A     |   'PRODP'       2: |          |
        |----------|  ----------->      |----------|
     1: |    B     |                 1: |    AB    |
        |----------|                    |----------|
```

'PRODP': (Checksum: # 51299d, Size: 202.5 bytes)

```
«  DUP   SIZE   1   GET
   →   a   da
   «   DUP   DUP   0   CON   DUP   SIZE   1   GET   DUP
       da   +   1   –   1   →LIST
       →   b   c   db   dim
       «   { 1 }   db   +   RDM
           1   db   START
                a   OBJ→   DROP   c   OBJ→   DROP
           NEXT
           db   DROPN
           db   dim   +   →ARRY   *   dim   RDM
       »
   »
»
```

Example: (in 1 second)

```
   |                     |
   |---------------------|
2: | [ 1  5  -9  7  2 ] |   'PRODP'
   |---------------------|  -------->
1: | [ 3  2  -7  -11 ]  |              1: | [3 17 -24 -43 28 54 -91 -22 ] |
   |---------------------|
```

since, if `P = X^4 +5X^3 -9X^2 +7X +2   et   Q = 3X^3 +2X^2 -7X -11,`

then:

`PQ = 3X^7 +17X^6 -24X^5 -43X^4 +28X^3 +54X^2 -91X -22`

POWERS OF A POLYNOMIAL

'POWP' calculates the nth power of a polynomial A (where n is a positive or zero integer) as shown in the functional diagram below:

```
    |          |                      |          |
2:  |    A     |     'POWP'       2:  |          |
    |          |   ---------->        |          |
1:  |    n     |                  1:  |    A     |
```

N.B: Program 'POWP' calls program 'PRODP'.

'POWP': (Checksum: # 7170d, Size: 168 bytes)

```
«  →  a  n
   «  1  DUP  →ARRY
      WHILE  n  0  >
      REPEAT
         IF  n  2  MOD  THEN  a  PRODP  END
         n  2  /  FLOOR  'n'  STO
         IF  n  THEN  a  DUP  PRODP  'a'  STO  END
      END
   »
»
```

Example:

We want to calculate (2X^2 -3X +7) ^5.

We therefore enter the vector [2 -3 7] at level 2, the integer 5 at level 1 and call 'POWP'.

The resultant vector is obtained in 3 seconds:

[32 -240 1280 -4440 12290 -25443 43015 -54390 54880 -36015 16807],

meaning that:

(2X^2 -3X +7)^5 = 32X^10 - 240X^9 + 1280X^8 .... - 36015X + 16807.

DIVISION OF TWO POLYNOMIALS (Euclidean division)

'DIV' performs a Euclidean division (i.e. in decreasing order of powers) of a polynomial A by a non-zero polynomial B.
This division is written A = BQ + R, where Q is the quotient of the division and R is the remainder (the degree of R is always less than that of B).

This gives us the following functional diagram:

```
2: |  A  |    'DIV'      2: |  Q  |
          ------------>
1: |  B  |               1: |  R  |
```

N.B: program 'DIV' calls program 'ELML'.

'DIV': (Checksum: # 24501d, Size: 379 bytes)

```
«  ELML  DUP  1  GET
   «  DUP  IM  ABS  IF  NOT  THEN  RE  END  »
   →  d  real
   «  DUP  SIZE  1  GET  ROT  DUP  SIZE  1  GET
      →  b  tb  a  ta
      «  a
         IF  tb  1  ==  THEN  d  /  0  1  →ARRY
            ELSE  IF  ta  tb  <  THEN  0  1  →ARRY  SWAP
               ELSE
                  ta  tb  FOR  i
                     DUP  1  GET  d  /  SWAP  OVER
                     b  i  {}  +  RDM  *  -  OBJ→
                     1  GET  1  -  →ARRY  SWAP  DROP
                  -1  STEP
                  ELML
                  →  r
                  «  ta  tb  -  1  +  →ARRY  real  EVAL  r
                  »
                  real  EVAL
               END
            END
      »
   »
»
```

Example: (in 3 seconds)

```
2: | [ 1 2 3 4 5 6] |    'DIV'      2: | [ 1 1 1 2 ] |
                      ------------>
1: |   [ 1 1 1 ]    |               1: |   [ 2 4 ]   |
```

Meaning that for the division of polynomial `A = X^5 +2X^4 +3X^3 +4X^2 + 5X +6` by polynomial `B = X^2 +X +1`, the quotient is:
`Q = X^3 +X^2 +X +2`, and the remainder is `R = 2X+4`.

DIVISION OF TWO POLYNOMIALS IN INCREASING ORDER OF POWERS

'DIVIP' divides a polynomial A by a polynomial B (whose constant coefficient is non-zero) in increasing order of powers up to n .

This division is written:

$$A = BQ + X^{n+1}\ R$$

where Q is the quotient (whose degree is less than or equal to n) and R is the remainder (the polynomial $X^{n+1}\ R$ is also called the remainder).

This gives the following functional diagram:

3:	A		3:	
2:	B	'DIVIP' ——>	2:	Q
1:	n		1:	R

NOTE:

Unlike in most of the programs in the 'POLY' directory, the polynomials A and B must be represented by the vector of their coefficients in increasing order of powers of the unknown. For example, `[ 1 5 -4 7 ]` represents `1 +5 -4X^2 +7X^3`. The polynomials Q and R are obtained in the same format.

N.B: Program 'DIVIP' calls program 'DIV'.

'DIVIP': (Checksum: # 6051d, Size: 53.5 bytes)

```
«   OVER   SIZE   1     GET    +   1     →LIST
    ROT    SWAP   RDM   SWAP       DIV
»
```

Example: (in 3 seconds)

3:	[1 2 3 4]		3:	
2:	[1 2 1]	'DIVIP' ——>	2:	[1 0 2 0 -2 4]
1:	5		1:	[-6 -4]

Meaning that:

`1 +2X +3X^2 +4X^3 = (1 +2X +X^2)(1 +2X^2 -2X^4 +4X^5) + X^6(-6-4X)`

COMPOSITION OF TWO POLYNOMIALS

===================================

'COMP' calculates the composite P(Q) of two polynomials P and Q as shown in the functional diagram below:

```
    |            |                    |            |
    |------------|                    |------------|
 2: |     Q      |      'COMP'     2: |            |
    |------------|    ---------->     |------------|
 1: |     P      |                 1: |    P(Q)    |
    |------------|                    |------------|
```

N.B: 'COMP' calls programs 'PRODP' and 'REV'

'COMP': (Checksum: # 53645d, Size: 125 bytes)

```
«  REV    SWAP
   →   p
   «   IF    DUP    SIZE    {1}    ≠    THEN
             OBJ→    1    GET    SWAP    1    →ARRY
             2    ROT    START
                  p   PRODP    DUP    SIZE
                  DUP2    GET    4    ROLL    +    PUT
             NEXT
       END
   »
»
```

Example: (in 2 seconds)

```
    |               |
    |---------------|
 2: | [ 1 5 -2 3 ]  |      'COMP'
    |---------------|    ---------->      |                              |
 1: | [ 1  1  1 ]   |                  1: | [ 1 10 21 -13 39 -14 13 ]    |
    |---------------|                     |------------------------------|
```

Meaning that if we assume that:

`P(X) = X^2 +X +1` and `Q(X) = X^3 +5X^2 -2X +3`, then

`P(Q(X)) = Q(X)^2 +Q(X) +1`

`= X^6 + 10X^5 +21X^4 -13X^3 +39X^2 -14X +13`

TRANSLATION OF A POLYNOMIAL

'TRNS' translates a polynomial `x -> P(x)` into the polynomial `x -> Q(x)=P(x+a)`, as shown in the functional diagram below:

2:	P(x)	'TRNS'	2:	
1:	a	----->	1:	Q(x)=P(x+a)

'TRNS': (Checksum: # 40631d, Size: 145.5 bytes)

```
«  OVER   SIZE   1   GET
   →   a   n
   «  IF   n   1   ≠   THEN
              n   2   FOR   i
                  2   i   FOR   j
                      DUP   j   GET   OVER   j   1   -
                      GET   a   *   +   j   SWAP   PUT
                  NEXT
              -1   STEP
      END
   »
»
```

Example: (in less than 2 seconds)

2:	[2 -1 0 3 5]	'TRNS'	2:	
1:	3	----->	1:	[2 23 99 192 149]

since, if `P(X) = 2X^4 -X^3 +3X +5 ,`
then `Q(X)= P(X+3) = 2X^4 +23X^3 +99X^2 +192X +149`

Note:

Program 'TRNS' also gives the decomposition of the polynomial `x -> P(x)` as a function of the powers of `(x-a)`

The example above can thus be interpreted by saying that:

if `P(X) = 2X^4 -X^3 +3X +5,`
then `P(X) = 2(X-3)^4 +23(X-3)^3 +99(X-3)^2 +192(X-3) +149`

We can also say that 'TRNS' changes the variable `y=(x-a)` in the polynomial `P(x)` and that the result is given as a polynomial of the variable y.

ROOTS OF A 2ND DEGREE POLYNOMIAL

'DEG2' calculates the two real or complex roots a and b of a 2nd degree polynomial p, with real or complex coefficients. This gives the following functional diagram:

```
2: |        |     'DEG2'      2: |   a    |
1: |   P    |  ---------->    1: |   b    |
```

'DEG2' also calculates the root a of a 1st degree polynomial as shown in the functional diagram below:

```
                  'DEG2'
1: |   P    |  ---------->    1: |   a    |
```

N.B: 'DEG2' calls program 'ELML'.

'DEG2': (Checksum: # 60963d, Size: 354 bytes)

```
«  ELML    DUP     SIZE
   →   p     s
   «   IF    s    {3}     ==     THEN
             'p(2)^2 - 4*p(1)*p(3)'    EVAL    √
             →    rd
             «    '-(p(2) + rd)/2/p(1)'     EVAL
                  '( -p(2) + rd)/2/p(1)'    EVAL
             »
         ELSE
             IF    s    {2}    ==    THEN
               p     OBJ→    DROP    SWAP
               IF    DUP    THEN    /    NEG    ELSE    DROP2    END
             END
       END
   »
»
```

Examples: (results obtained within 1 second)

```
2: |              |     'DEG2'      2: |        -2          |
1: | [ 1  3  2 ]  |  ---------->    1: |        -1          |

2: |              |     'DEG2'      2: |(-.5,-.866025403785)|
1: | [ 1  1  1 ]  |  ---------->    1: | (-.5,.866025403785)|
```

In 5 FIX mode:

```
2: |                            |    'DEG2'     2: | ( -0.21392 , 2.48243 )  |
1: | [ (1,1) (2,-1) (3,2) ]     |  -------->    1: | ( -0.28608 , -0.98243 ) |
```

ROOTS OF A 3RD DEGREE POLYNOMIAL

'DEG3' calculates the three real or complex roots a, b and c of a 3rd degree polynomial p, with real or complex coefficients. This gives the following functional diagram:

```
   |            |                      |            |
3: |            |                   3: |     a      |
2: |            |       'DEG3'      2: |     b      |
1: |     P      |    ---------->    1: |     c      |
```

N.B: Those of the roots a, b or c that are real are however given in complex form (with a zero or virtually zero imaginary part owing to round-off errors).

'DEG3': (Checksum: # 23540d, Size: 404 bytes)

```
«  DUP   1   GET   /   OBJ→   DROP
   →  a  b  c
   «  DROP  'b - a^2/3'  EVAL  3  /
      '2*a^3/27 - a*b/3+c'  EVAL  2  /
      →  p  q
      «  IF  p  ABS  THEN
               q  p  √  p  *  /  NEG  ASINH
               →  z
               «  0  2  FOR  k
                     '2*√p*SINH((z+2*i*k*π)/3)-a/3'  →NUM
                  NEXT
               »
            ELSE
               0  2  FOR  k
                  '(-2*q)^(1/3)*EXP(2*i*k*π/3)-a/3'  →NUM
               NEXT
         END
      »
   »
»
```

Example: (in 5 FIX mode for the result, in 2 seconds)

```
   |                 |                    |                       |
3: |                 |                 3: |  ( 0.19326, 0.42356 ) |
2: |                 |     'DEG3'      2: |  ( 0.19326,-0.42356 ) |
1: | [ 1 -5 2 -1 ]   |  ---------->    1: |( 4.61347, 7.48801E-12 )|
```

In this particular case, the polynomial P has 2 complex conjugate roots and a real root approximately equal to 4.61347.

ROOTS OF A 4TH DEGREE POLYNOMIAL

'DEG4' calculates the four real or complex roots a, b, c and d of a 4th degree polynomial p, with real or complex coefficients. This gives the following functional diagram:

4:			4:	a
3:			3:	b
2:		'DEG4'	2:	c
1:	P	———>	1:	d

N.B: 'DEG4' calls programs 'DEG2' and 'DEG3'. The real roots are given in complex form, with a zero imaginary part (taking round-off errors into account).

'DEG4': (Checksum: # 58151d, Size: 630 bytes)

```
«  DUP   1   GET   /   OBJ→   DROP
   →  a  b  c  d
   «  DROP  1  b  NEG  'a*c -4*d'  EVAL  'd*(4*b -a*a) -c*c'  EVAL
      4  →ARRY  DEG3
      3  →ARRY  DUP  {3}  'a*a/4 -b'  EVAL  CON  +  1
      →  t  j
      «  IF  'ABS(t(2)) > ABS(t(1))'  THEN  2  'j'  STO  END
         IF  'ABS(t(3)) > ABS(t(j))'  THEN  3  'j'  STO  END
             t  j  GET  √  SWAP  j  GET
             →  w  y
             «  IF  w  (0,0)  ==  THEN
                       a  4  /  NEG  DUP  DUP2
                    ELSE
                       0  w  'a*y/2 -c'  EVAL  w  /  2  /  3
                       →ARRY  1  a  2  /  y  2  /  3  →ARRY
                       →  v1  v2
                       «  v1  v2  +  DEG2
                          v1  v2  -  DEG2
                       »
                END
             »
      »
   »
»
```

Example: (in 5 seconds)

			4:	(-7 , 0)
3:			3:	(-1.9999999998 , -3)
2:		'DEG4'	2:	(11 , 0)
1:	[1 0 -80 -360 -1001]	———>	1:	(-2 , 3)

Here the roots of P are -7, 11, -2-3i and -2+3i.

ROOTS OF A POLYNOMIAL USING BAIRSTOW'S METHOD

'BRST' calculates an approximate value of the real or complex roots $a_1, a_2, a_3, \ldots, a_n$ of an nth degree polynomial A(x) with real or complex coefficients (if $n<5$ 'BRST' simply calls one of the programs 'DEG2', 'DEG3' or 'DEG4'). The functional diagram is as shown below:

			n:	a_n
4:			.	.....
3:			3:	a_3
2:		'BRST'	2:	a_2
1:	A	——————>	1:	a_1

N.B: 'BRST' calls programs 'DEG2', 'DEG3', 'DEG4' and 'ELML'.

The principle is as follows:

We want to find two complex numbers s and p such that A(x) will be divisible by x^2-sx-p. This requires us to use Newton's iterative algorithm to find s and p.

We therefore start with an initial value $[s_0, p_0]$ then build a sequence $[s_n, p_n]$ which should converge towards a solution [s, p].

If [s, p] is found, the polynomial A can be written A(x)=(x^2-sx-p)B(x), where B is an n-2 degree polynomial. We then enter the two roots of x^2-sx-p in the stack (using 'DEG2') and apply the same procedure for the polynomial B.

The calculation sequence stops as soon as the degree of the polynomial obtained is less than or equal to 4, in which case it is best to call 'DEG2', 'DEG3' or 'DEG4'.

'BRST' halts whenever two new roots of A are entered in the stack. The user then presses CONT to search for other roots.

When 'BRST' halts, we can modify the default value of the variable **'eps'**, which is 1E-6 by default and which alters the stage at which iterations are stopped, and the default value of the variable **'max'**, which is 20 (the maximum number of iterations before the process will be considered to be divergent).

The process may in fact diverge (or converge slowly). This can be seen when the program halts without two new roots having been entered in the stack. We can then modify the variable **'v'**, which is the current value of $[s_n, p_n]$ (and is a **vector** of two elements), before resuming the iterative sequence via CONT. We can also resume the iterative sequence without modifying anything at all.

Important note: if you modify variables while the program 'BRST' is halted, you must be careful to ensure that the stack is as it was when the program was halted before pressing CONT. Small letters must be used for the variables 'eps', 'max' and 'v'.

TEXT OF PROGRAM 'BRST'

'BRST': (Checksum: # 53898d, Size: 726 bytes)

```
«   DUP   SIZE   1   GET   0   0   0   0   0   0   .000001   20
    →   a   n   v   s   p   b   d   k   eps   max

    «   IF   n   6   ≥   THEN

            1   DUP   2   →ARRY   'v'   STO

            DO
                0   'k'   STO

                DO
                    v   OBJ→   DROP   'p'   STO   's'   STO   0
                    a   1   GET
                    2   n   FOR   i
                        DUP   s   *   3   PICK   p
                        *   +   a   i   GET   +
                    NEXT
                    n   →ARRY   'b'   STO   DROP   v   0   0
                    b   1   GET
                    2   n   1   -   FOR   i
                        ROT   DROP   DUP   s   *   3   PICK
                        p   *   +   b   i   GET   +
                    NEXT
                    SWAP   DUP   4   ROLLD   { 2   2 }   →ARRY
                    INV   b   n   1   -   GET   b   n   GET   2   →ARRY   *
                    DUP   RNRM   v   RNRM   /   'd'   STO
                    -   'v'   STO   'k'   1   STO+
                UNTIL   d   eps   <   k   max   ≥   OR   END

                IF   k   max   <   THEN
                    1   s   NEG   p   NEG   3   →ARRY
                    DEG2   'n'   2   STO-
                    b   n   1   →LIST   RDM   'a'   STO
                END

                HALT

            UNTIL   n   6   <   k   max   <   AND   END

        END

        a   ELML   "DEG"   OVER   SIZE   1   GET   1   -
        2   MAX   →STR   +   1   4   SUB   OBJ→
    »
»
```

PRACTICAL EXAMPLE USING PROGRAM 'BRST'

We want to find the roots of the polynomial:

`P(X)= X^10 +2X^9 +3X^8 +4X^7 +5X^6 +6X^5 +7X^4 +8X^3 +9X^2 +10X +11.`

We therefore enter the vector `[ 1 2 3 4 5 6 7 8 9 10 11 ]` at level 1 of the stack and call 'BRST'.

After 32 seconds, we obtain two complex conjugate roots of P in the stack , i.e.:

$a_1 =$ (-1.26463096509, -.357261654484)
and $a_2 =$ (-1.26463096509, .357261654484)

We then press CONT and after 21 seconds, we find two new complex conjugate roots:

$a_3 =$ (.442765764928, -1.17374073066)
and $a_4 =$ (.442765764928, 1.17374073066)

Pressing CONT again, we find another two complex conjugate roots after 23 seconds:

$a_5 =$ (-.246722626138, -1.26288540248)
and $a_6 =$ (-.246722626138, 1.26288540248)

Pressing CONT again, after 7 seconds we obtain the last four roots of P, which are both pairs of complex conjugates. The program 'BRST' is now terminated. The last four roots are obtained by 'DEG4' (called by 'BRST'). These roots are:

$a_7 =$ (-.88465843109, -.959966681655)
$a_8 =$ (-.88465843109, .959966681655)
$a_9 =$ (.95324625739, .72511051529)
and $a_{10} =$ (.95324625739, -.72511051529)

It is important to be able to control the accuracy of results obtained.
In the case of real roots (if there are any), we can improve the accuracy by transforming the polynomial P (written in vector form) into conventional algebraic notation, i.e. `'X^10 +2*X^9 +3*X^8... +9*X^2 +10*X +11'` (using the program V→P) and using the program SOLVR, starting with the approximate root obtained with 'BRST'.

In the general case, we can use the approximation (where a is an exact root of P and a is the approximate value found, and provided that a is a simple root):

$$| P(a) - P(\overline{a}) | \approx | a - \overline{a} | \; | P'(\overline{a}) |$$

and therefore:

$$| a - \overline{a} | \approx | P(\overline{a}) | \; / \; | P'(\overline{a}) |$$

For this calculation, we use the programs 'DERIV' (derivative of P) and 'VALP' (calculation of values of P and P' in a).
For a_{10}, we therefore find $| a_{10} - \overline{a}_{10} | \approx 3.18E-9$ and therefore:

$| a_{10} - \overline{a}_{10} | \leq 5E-9$

For a_8, we find $| a_8 - \overline{a}_8 | \approx 6.01E-9$
For a_6, we find $| a_6 - \overline{a}_6 | \approx 9.05E-9$
For a_4, we find $| a_4 - \overline{a}_4 | \approx 6.33E-9$
For a_2, we find $| a_2 - \overline{a}_2 | \approx 1.48E-10$

VALUE OF A POLYNOMIAL AT A POINT

'VALP' calculates the value of the polynomial P at a point x, as shown in the functional diagram below:

```
2: |   P   |    'VALP'     2: |        |
           |   -------->      |        |
1: |   x   |               1: |  P(x)  |
```

N.B: program 'VALP' calls program 'REV'.

'VALP': (Checksum: # 47419d, Size: 66.5 bytes)

```
«  →  x
   «  REV  OBJ→  1  GET  0  1  ROT
      START  x  *  +  NEXT
   »
»
```

Example: (in less than one second)

```
2: [ 4 -5 6 3 -2 0 9 ]   'VALP'   2:
                        ------->
1:        1.57                    1:  64.3440621073
```

We can also use the HP48's EVAL function by entering:
`'4*X^6 -5*X^5 +6*X^4 +3*X^3 -2*X^2 +9'` at level 1, then `1.57` for `X` and calling EVAL. EVAL is quicker than 'VALP', but this does not make up for the fact that the polynomial has to be entered in algebraic form.

REVERSING THE ORDER OF THE COMPONENTS OF A VECTOR

'REV' reverses the order of the components of a vector at level 1 of the stack. For example (in less than one second):

```
                      'REV'
1: [ 1 2 3 4 5 6]   -------->   1: [ 6 5 4 3 2 1]
```

'REV': (Checksum: # 55757d, Size: 67.5 bytes)

```
«  OBJ→  1  GET
   →  n
   «  1  n  FOR  i  i  ROLL  NEXT  n  →ARR
Y
   »
»
```

'REV' is useful when writing a polynomial in increasing or decreasing order of powers (program 'DIVIC').

DERIVATIVE OF A POLYNOMIAL

'DERIV' calculates the derivative polynomial P' of P, as shown in the functional diagram below:

```
       |-------------------|      'DERIV'          |-------------------|
    1: |         P         |   ------------->   1: |         P'        |
```

'DERIV': (Checksum: # 27163d, Size: 113.5 bytes)

```
«  OBJ→   1   GET   1   -
   →  n
   «  DROP
      IF   n   THEN
         1   n   FOR  i   i   *   n   ROLLD   NEXT
n
      ELSE   0   1   END
      →ARRY
   »
»
```

Example: (in less than one second)

```
    1: [ 3 -2 8 1 0 -6 4 ]     'DERIV'     1: [ 18 -10 32 3 0 -6]
                             ------------>
```

Note:
This method is much quicker than entering:

```
    2: '3*X^6-2*X^5+8*X^4+X^3-6*X+4'
    1:              'X'
```

then doing '∂', which gives the following after 5 seconds:

```
    1:  '3*(6*X^5)-2*(5*X^4)+8*(4*X^3)+3*X^2-6'
```

We then have to go to the ALGEBRA menu and press COLCT, which gives the following after 8 seconds:

```
    1:  '-6+3*X^2+32*X^3-10*X^4+18*X^5'
```

PRIMITIVE OF A POLYNOMIAL

'PRIM' calculates the primitive Q cancelling to 0 of the polynomial P, as shown in the functional diagram below:

```
1: [    P    ]   'PRIM'   ->   1: [    Q    ]
```

'PRIM': (Checksum: # 61049d, Size: 83 bytes)

```
«  OBJ→   1   GET
   →   n
   «   1   n   FOR   i   i   /   n   ROLLD   NEXT
       0   n   1   +   →ARRY
   »
»
```

Example: (in less than one second)

```
1: [12 -1 3 6 -1 5 ]   'PRIM'   ->   1: [ 2 -.2 .75 2 -.5 5 0 ]
```

INTEGRAL OF A POLYNOMIAL OVER A SEGMENT

'INTP' calculates the integral of the polynomial P from a to, as shown in the functional diagram below:

```
3: [    P    ]
2: [    a    ]   'INTP'   ->
1: [    b    ]                 1: [ ∫a^b P(x) dx ]
```

N.B: Program 'INTP' calls programs 'PRIM' and 'VALP'.

'INTP': (Checksum: # 28489d, Size: 74 bytes)

```
«  →   a   b
   «   PRIM   DUP   b   VALP   SWAP   a   VALP   -
   »
»
```

Example: (in one second)

```
3: [ 5 3 2 -8 4 ]            3: [                ]
2: [     -3     ]   'INTP'   2: [                ]
1: [      5     ]   ->       1: [ 3845.33333335  ]
```

GCD OF TWO POLYNOMIALS

'GCDP' calculates the GCD (greatest common divisor) of two polynomials A and B, as shown in the functional diagram below:

```
2: |  A  |    'GCDP'    2: |            |
1: |  B  |  -------->   1: | Gcd( A, B) |
```

The polynomial thus obtained is unitary (i.e. whose highest-degree term = 1).
N.B: program 'GCDP' calls program 'DIV'.

'GCDP': (Checksum: # 54781d, Size: 65 bytes)

```
«   WHILE   DUP   ABS
    REPEAT
        SWAP   OVER   DIV   SWAP   DROP
    END
    DROP   DUP   1   GET   /
»
```

Example: (in 4 seconds)

```
2: | [ -2 5 8 -13 1 -5 ] |    'GCDP'    2: |                  |
1: |   [ 2 5 3 -7 -15 ]  |  -------->   1: | [ 1 .5 -2.5 ]    |
```

(in this example the coefficient .5 of the gcd is obtained with a round-off error of 2E-12).

LCM OF TWO POLYNOMIALS

'LCMP' calculates the lcm (least common multiple) of two polynomials A and B, as shown in the functional diagram below:

```
2: |  A  |    'LCMP'    2: |            |
1: |  B  |  -------->   1: | Lcm( A, B) |
```

The polynomial thus obtained is unitary (whose highest- degree term = 1).
N.B: program 'LCMP' calls programs 'CGDP', 'DIV' and 'PRODP'.

'LCMP': (Checksum: # 28888d, Size: 56 bytes)

```
«   DUP2
    GCDP   DIV   DROP   PRODP   DUP   1   GET   /
»
```

Example: (in 6 seconds)

```
2: | [ 5 17 7 3 ]  |    'LCMP'    2: |                             |
1: | [ 2 5 1 12 ]  |  -------->   1: | [ 1 2.9 1.7 6.7 2.5 1.2]    |
```

SWITCHING FROM VECTOR FORM TO ALGEBRAIC FORM

'V→P' transforms a vector (representing a polynomial P written in decreasing order of powers) into an algebraic expression. This gives the following functional diagram:

```
    |             |                    |             |
    |-------------|                    |-------------|
 2: |   vector    |   'V → P'      2:  |             |
    |-------------|  ---------->       |-------------|
 1: | expression1 |                1:  | expression2 |
    ---------------                    ---------------
```

where "vector" is the vector representing the polynomial P, "expression1" is the algebraic expression replacing the unknown of the polynomial P and "expression2" is the algebraic expression of the polynomial thus obtained.
If, for example, "expression1" is equal to 'X', we obtain the polynomial P written in conventional algebraic form.

'V→P': (Checksum: # 44891d, Size: 117.5 bytes)

```
«   →    v
    «    DUP    SIZE    {}    +    1    GET
         →    p    n
         «    0    1    n    FOR    i
                   p    i    GET    v    n    i    -    ^    *    +
              NEXT
         »
    »
»
```

Example: (in 2 seconds)

```
    |                     |                    |                        |
    |---------------------|                    |------------------------|
 2: | [ 7 0 -4 1 2 0 ]    |  'V → P'     2:    |                        |
    |---------------------|  -------->         |------------------------|
 1: |        'X'          |              1:    | '7*X^5-4*X^3+X^2+2*X'  |
    -----------------------                    --------------------------
```

We can replace 'X' with other algebraic expressions like, for example, `'Y'`, `'x'`, `'COS(T)'`, etc. In the latter case, the polynomial above is written:

`'7*COS(T)^5-4*COS(T)^3+COS(T)^2+2*COS(T)'`

'V→P' is useful if we want to display a polynomial more easily in vector form (when obtained as a result of one of the programs in the directory, for example) or when using the calculator's SOLVR or DRAW programs.

'V→P' will also work if the argument at level 2 is a list. We can thus use it in conjunction with program 'T→Q' in the 'R.C' directory.

TCHEBYSHEV POLYNOMIALS

'TCHEB' calculates Tchebyshev polynomials of the first kind T_n and the second kind U_n.

Polynomials T_n are defined by:
$T_0(x)=1$, $T_1(x)=x$, and where $n \geq 2$, $T_n(x)=2xT_{n-1}(x) - T_{n-2}(x)$.

Polynomials Un are defined by:
$U_0(x)=1$, $U_1(x)=2x$, and where $n \geq 2$, $U_n(x) = 2xU_{n-1}(x) - U_{n-2}(x)$.

This gives the following functional diagram:

* to obtain Tchebyshev polynomials of the first kind:

```
2: | n |      'TCHEB'      2: |   |
1: | 1 |   ------------>   1: | T |
```

* to obtain Tchebyshev polynomials of the second kind:

```
2: | n |      'TCHEB'      2: |   |
1: | 2 |   ------------>   1: | U |
```

'TCHEB': (Checksum: # 24506d, Size: 231.5 bytes)

```
«  IF  {1 2}  OVER  POS  THEN  0  2  →ARRY  1  DUP  →ARRY
   →  n  b  a
   «  IF  n  THEN
         IF  n  1  ==  THEN  b  ELSE
            3  n  1  +  FOR  k
               b  DUP  k  1  →LIST  RDM  2  *  0  0
               a  V→  k  →ARRY  -  'b'  STO  'a'  STO
            NEXT  b
         END
      ELSE  a  END
   »
   END
»
```

Example: (in less than two seconds)

```
2: | 5 |      'TCHEB'      2: |                     |
1: | 1 |   ------------>   1: | [ 16 0 -20 0 5 0 ]  |
```

As T (X) = 16*X^5 -20*X^3 +5*X

CALCULATING A MATRIX POLYNOMIAL

'PMAT' allows you to apply a polynomial P to a square matrix M, so as to obtain the matrix P(M). The coefficients of polynomial P and/or matrix M can be either real or complex.

```
      ┌─────────────┐                 ┌─────────────┐
   2: │  Matrix M   │     'PMAT'   2: │             │
      ├─────────────┤   ---------->   ├─────────────┤
   1: │ Polynomial  │              1: │ Matrix P(M) │
      └─────────────┘                 └─────────────┘
```

If $P(x) = ax^n + bx^{n-1} + \ldots + cx + d$, then:

$P(M) = aM^n + bM^{n-1} + \ldots + cM + dI$, where I is the identity matrix with the same format as M

NB: program 'PMAT' calls program 'REV'

'PMAT': (Checksum: # 12277d, Size: 97 bytes)

```
« SWAP DUP IDN
  → m id
  « REV OBJ→ 1 GET m 0 CON
    1 ROT START
      m * SWAP id * +
    NEXT
  »
»
```

Example: (in 2 seconds)

```
      ┌───────────────────┐
      │ [[  1  0 -2  ]    │
      │  [ -3  1  4  ]    │     'PMAT'          ┌──────────────────────────┐
   2: │  [  0  5  1  ]]   │                     │ [[  35   -20   -46  ]    │
      ├───────────────────┤   ---------->       │  [ -69    75   104  ]    │
   1: │  [ 1 -1 2 3  ]    │                  1: │  [ -30   115    75  ]]   │
      └───────────────────┘                     └──────────────────────────┘
```

EXPANDING A PRODUCT OF POWERS OF POLYNOMIALS

'EXPPP' allows you to expand a product of powers of a polynomial, i.e. polynomials written:

$$A = B_1^{\alpha_1} \quad B_2^{\alpha_2} \quad \ldots\ldots \quad B_n^{\alpha_n}.$$

The factorized form must be entered in the form of a list compiled as follows:

List = { B_1 α_1 B_2 α_2 ..... B_n α_n }

All exponents, even if equal to 1, must be included.

The functional diagram can therefore be written:

```
                          'EXPPP '
1: [      List      ]  ------------>  1: [       A        ]
```

where A is the polynomial (written in vector form) obtained from the expansion.

N.B: 'EXPPP' calls programs 'POWP' and 'PRODP'.

'EXPPP': (Checksum: # 60982d, Size: 87 bytes)

```
«  1    DUP    →ARRY
   1    3   PICK   SIZE   FOR   i
        OVER   i   GETI   3   ROLLD   GET   POWP   PRODP
   2    STEP
   SWAP    DROP
»
```

Example: (in five seconds)

```
1: { [ 1 5 ] 2 [ 1 -2 4 ] 3 }

'EXPPP'
------->   1: [1 4 -11 34 136 -536 1504 -1760 1600
```

In other words: `(X+5)^2*(X^2-2X+4)^3` is equal to:

`X^8 + 4X^7 -11X^6 + 34X^5 + 136X^4 -536X^3 + 1504X^2 -1760X + 1600`

PRIMITIVE OF P(x)cos(ax)+Q(x)sin(ax)

'PPCS' calculates a primitive, in symbolic form, of a function written P(x)cos(ax)+Q(x)sin(ax), where P and Q are both polynomials and a is a non-zero real number.
This gives the following functional diagram:

3:	P		3:	
2:	Q	'PPCS' ——>	2:	
1:	a		1:	primitive

Here the polynomials P and Q are written in their usual format (as vectors of the components in decreasing order of powers) and "primitive" is an algebraic expression representing a symbolic primitive of the application `P(x)cos(ax)+Q(x)sin(ax)`. This primitive is obtained in the form `A(x)cos(ax)+B(x)sin(ax)`, where A and B are both polynomial expressions in terms of the variable 'X'.

N.B: 'PPCS' calls programs 'DERIV', 'ADDP' and 'V→P'.

'PPCS': (Checksum: # 26977d, Size: 311 bytes)

```
«  →  p  q  a
   «  p  DERIV  q  a  *  NEG  ADDP  DUP  SIZE  1  GET
      →  r  n
      «  0  0  n  1  FOR  k
               r  n  k  -  1  +  GET  3  PICK
               k  1  +  *  k  *  -  a  SQ  /
         -1  STEP
         n  →ARRY  3  ROLLD  DROP2  DUP  'X'  V→P  a
         'X'  *  COS  *  p  ROT  DERIV  NEG  ADDP  a
         /  'X'  V→P  a  'X'  *  SIN  *  +
      »
   »
»
```

Example: (in 5 seconds)

3:	[1 -1 2 3]		2:	
2:	[2 -5 1]	'PPCS' ——>		'(8*X^2+2*X-58)*COS(.5*X)+(2*X^3-2*X^2-28*X+2)*SIN(.5*X)'
1:	.5		1:	

Indeed: $\int$ `[ (X^3-X^2+2X+3)cos(X/2) + (2X^2-5X+1)sin(X/2) ] dX`

is equal to `(8X^2+2X-58)cos(X/2)+(2X^3-2X^2-28X+2)*sin(X/2)` (to the nearest integration constant).

PRIMITIVE OF P(x)exp(ax)

'PPEX' calculates a primitive, in symbolic form, of a function written `P(x)exp(ax)`, where P is a polynomial and a is a non-zero real number.

This gives the following functional diagram:

```
         +-----------+                      +-----------+
   2:    |     P     |     'PPEX'     2:    |           |
         +-----------+   ---------->        +-----------+
   1:    |     a     |                1:    | primitive |
         +-----------+                      +-----------+
```

Here the polynomial P is written in the usual format (as a vector of the components in decreasing order of powers) and "primitive" is an algebraic expression representing a symbolic primitive of the application `P(x)exp(ax)`. This primitive is obtained in the form `A(x)exp(ax)`, where A is a polynomial expression in terms of the variable 'X'.

N.B: 'PPEX' calls program 'V→P'.

'PPEX': (Checksum: # 44523d, Size: 163 bytes)

```
«  OVER   SIZE   1   GET
   →   p   a   n
   «   0   n   1   FOR   k
             p   n   k   -   1   +   GET   OVER   k   *   -   a   /
       -1   STEP
       n   →ARRY   SWAP   DROP   'X'   V→P   a   'X'   *   EXP   *
   »
»
```

Example: (in two to three seconds)

```
         +---------------+                    +------------------------+
   2:    | [ 1 0 0 0 0 ] |    'PPEX'          |                        |
         +---------------+   -------->        +------------------------+
   1:    |      -1       |               1:   | '(-X^4-4*X^3-12*X^2-   |
         +---------------+                    |  24*X-24)*EXP(-X)'     |
                                              +------------------------+
```

Indeed: $\int$ `X^4 exp(-X) dX = (-X^4-4X^3-12X^2-24X-24)exp(-X)`

(to the nearest integration constant).

SOLVING THE EQUATION AU + BV = C

Program 'ABCUV' allows you to obtain the best possible solution (U,V) (i.e. the solution that keeps the degrees of U and V to a minimum) to the equation `AU+BV=C`, where A, B and C are three given polynomials and U and V are two unknown polynomials.

For such an equation to be satisfied by at least one solution (and it can be satisfied by an infinite number of solutions) it is necessary and sufficient that the polynomial C be a multiple of the GCD of the polynomials A and B.

This gives the following functional diagram:

3:	A		3:	
2:	B	'ABCUV' ——>	2:	U
1:	C		1:	V

where the polynomials A and B are written in their usual form (vector of the components in decreasing order of powers).

If there is no solution to the equation `AU+BV=C`, the program is terminated by the message "No solution".

N.B: Program 'ABCUV' calls programs 'DIV', 'PRODP', 'ADDP' and 'ELML'.

'ABCUV': (Checksum: # 6634d, Size: 358 bytes)

```
«   0   1   →ARRY   1   DUP   →ARRY   DUP2
    →   a   b   c   u   v   y   x
    «   WHILE   a   b   DIV   DUP   ABS
        REPEAT
            b   'a'   STO   'b'   STO
            u   DUP   3   PICK   PRODP   NEG
            x   ADDP   'u'   STO   'x'   STO
            v   DUP   ROT   PRODP   NEG
            y   ADDP   'v'   STO   'y'   STO
        END
        DROP2   c   b   DIV
        IF   ABS   THEN
            DROP   "No solution"
        ELSE
            u   OVER   PRODP   ELML
            v   ROT   PRODP   ELML
        END
    »
»
```

PRACTICAL EXAMPLES USING PROGRAM 'ABCUV'

Example 1: (computation time = 8 seconds, in 5 FIX mode)

3:	[1 -1 -2 3]		3:	
2:	[1 3 4]	'ABCUV' ——>		[-0.03681 0.00613]
1:	[1]		1:	[0.03681 -0.15337 .24540]

If we use 'T→Q' in the 'R.C' directory, we find the following coefficients:

* { '-6/163' '1/163' } at level 2.
* { '6/163' '-25/163' '40/163' } at level 1.

I.e. we obtain the result AU+BV=C with:

```
A = X^3 -X^2 -2X +3,     B = X^2 +3X + 4,    C=1,
U = (-6X^2 + 1)/163,     V = (6X^3 -25X^2 +40)/163
```

Example 2: (in 5 seconds)

3:	[1 -1 -1 -2]		3:	
2:	[1 1 -6]	'ABCUV' ——>	2:	
1:	[1 1]		1:	"No Solution"

This result means that the equation AU+BV=C with:
`A = X^3 -X^2 -X -2,    B = X^2 + X -6,    C = X + 1,`
has no solution.

The reason for this is that the GCD of the two polynomials A and B is the polynomial X-2, and C is not divisible by X-2.

ELIMINATING ZERO COEFFICIENTS TO THE LEFT OR THE RIGHT

'ELML' and 'ELMR' are two routines that eliminate zero coefficients at the start ('ELML') or at the end ('ELMR') of a vector V.

To avoid round-off errors, a coefficient is considered to be zero if its absolute value is less than 1E-5 (this value may be modified in the text of programs 'ELML' and 'ELMR').

The functional diagram of 'ELML' and 'ELMR' is as follows:

```
                          'ELML'
1: |      V      |   ------------->   1: |      W      |
                          'ELMR'
```

where W is the vector resulting from the truncation of V.

'ELML': (Checksum: # 8202d, Size: 134.5 bytes)

```
«   DUP    RNRM
    IF    .00001    <    THEN    DROP    0    1    →ARRY    ELSE
          OBJ→    1    GET    1    +
          WHILE    DUP    ROLL    DUP    ABS    .00001    <
          REPEAT    DROP    1    -    END
          OVER    ROLLD    1    -    →ARRY
    END
»
```

'ELMR': (Checksum: # 22420d, Size: 117 bytes)

```
«   DUP    RNRM
    IF    .00001    <    THEN    DROP    0    1    →ARRY
          ELSE    OBJ→    1    GET
          WHILE    OVER    ABS    .00001    <
          REPEAT    1    -    SWAP    DROP    END
          →ARRY
    END
»
```

<u>Example</u>:

The vector `V = [ 0 1E-11 1 -2 3 4 0 1E-7 0 ]` is transformed:
into `W = [ 1 -2 3 4 0 1E-7 0 ]` by 'ELML' and into
`W= [ 0 1E-11 1 -2 3 4 ]` by 'ELMR'.

RATIONAL FRACTIONS

The 'FRAC' directory must be installed as a sub-directory of the 'POLY' directory.

It contains programs written for rational fractions, i.e. functions written as the quotient R=A/B of two polynomial functions A and B.

'SMPF'	:	Simplifying a rational fraction.
'ADDF'	:	Addition of two rational fractions.
'PRODF'	:	Product of two rational fractions.
'F→Q'	:	Conversion of the coefficients of a rational fraction into rational numbers.
'V→F'	:	Writing a rational fraction in algebraic form
'VALF'	:	Value of a rational fraction at a point.
'POWF'	:	Powers of a rational fraction.
'DERVF'	:	Derivative of a rational fraction.

And of special importance:

'DECPF' : Decomposition of a rational fraction into partial fractions.

Most of the programs above require a rational fraction to be represented in the stack by superimposing two vectors, one representing the numerator and the other the denominator.

For example, the rational fraction; $R = \dfrac{X\hat{}3 - 2X\hat{}2 + 5X - 1}{11X\hat{}2 - 15X + 21}$

is represented in the stack by:

```
2: [ 1 -2  5 -1 ]
1:  [ 11 -15 21 ]
```

SIMPLIFYING A RATIONAL FRACTION

Program 'SMPF' simplifies (where possible) a rational fraction R=A/B to obtain a rational fraction C/D.

If the fraction A/B cannot be simplified it remains unchanged.

The functional diagram is as follows:

2:	A	'SMPF'	2:	C
1:	B	————>	1:	D

N.B: 'SMPF' calls programs 'GCDP' and 'DIV' in the 'POLY' directory.

'SMPF': (Checksum: # 5742d, Size: 94.5 bytes)

```
«   DUP2    GCDP
    IF     DUP     [ 1 ]    ≠     THEN
           ROT     OVER    DIV    DROP    3    ROLLD    DIV
    END
    DROP
»
```

Example: (in six seconds)

2:	[1 -2 -2 -3]	'SMPF'	2:	[1 1 1]
1:	[1 -1 -6]	————>	1:	[1 2]

In fact:

$$\frac{X^3 - 2X^2 - 2X - 3}{X^2 - X - 6} \quad \text{is equal to} \quad \frac{X^2 + X + 1}{X + 2}.$$

ADDITION AND PRODUCT OF RATIONAL FRACTIONS

Programs 'ADDF' and 'PRODF' let you add and multiply respectively two rational fractions A/B and C/D.
If we write the resulting rational fraction as E/F, the functional diagram is as follows:

4:	A		4:	
3:	B		3:	
2:	C	'ADDF'	2:	E
1:	D	——> or 'PRODF'	1:	F

N.B: 'PRODF' calls 'PRODP' in the 'POLY' directory.
'ADDF' calls 'PRODP', 'ELML', and 'ADDP'.

Note: no attempt to simplify is made after adding or multiplying.

'ADDF':(Checksum: # 41809d, Size: 111.5 bytes)

```
«  →  a  b  c  d
   «  a  d  PRODP  b  c  PRODP  ADDP  ELML
      b  d  PRODP
   »
»
```

'PRODF':(Checksum: # 14436d, Size: 46.5 bytes)

```
«  ROT  PRODP  3  ROLLD  PRODP  SWAP  »
```

Example: ('ADDF' in 3 seconds, 'PRODF' in 2 seconds).

4:	[1 2 1]		4:	
3:	[1 -3]		3:	
2:	[2 1 5]	'ADDF'	2:	[1 5 -1 5 -14]
1:	[1 1 1]	——>	1:	[1 -2 -2 -3]

4:	[1 2 1]		4:	
3:	[1 -3]		3:	
2:	[2 1 5]	'PRODF'	2:	[2 5 9 1 5]
1:	[1 1 1]	——>	1:	[1 -2 -2 -3]

CONVERTING THE COEFFICIENTS OF A RATIONAL FRACTION INTO RATIONAL NUMBERS

'F→Q' transforms the coefficients of a rational fraction A/B into a rational approximation. This is done by calling program 'A→Q' in the 'R.C' directory. The functional diagram looks like this:

2:	A	'F → Q'	2:	List A
1:	B	——————>	1:	List B

where "ListA" and "ListB" are the lists containing the rational approximations of the coefficients of the polynomials A and B.
N.B: 'F→Q' goes into the 'R.C' directory and calls 'A→Q' before returning to the 'FRAC' directory via the 'POLY' directory.

'F→Q':(Checksum: # 56371d, Size: 57 bytes)

```
«  R.C  SWAP  A→Q  SWAP  A→Q  POLY  FRAC  »
```

Example: (in 3 seconds)

2:	[2.125 3.5 1.875]	'F → Q'		{ '17/8' '7/2' '15/8' }
1:	[.225 -.025 .125]	——————>	1:	{ '9/40' '-1/40' '1/8'}

WRITING A RATIONAL FRACTION IN ALGEBRAIC FORM

'V→F' expresses a rational fraction A/B in terms of the variable 'X', with A and B given in vector form. This gives us the following functional diagram:

2:	A	'V → F'	2:	
1:	B	——————>	1:	Expression

N.B: 'V→F' calls 'V→P' in the 'POLY' directory.

'V→F':(Checksum: # 61852d, Size: 57 bytes)

```
«  SWAP  'X'  V→P  SWAP  'X'  V→P  /  »
```

Example: (in 2 seconds)

2:	[1 0 -2]	'V → F'	2:	
1:	[3 -1 5]	——————>	1:	'(X^2-2)/(3*X^2-X+5)'

VALUE OF A RATIONAL FRACTION AT A POINT

'VALF' calculates the value of the rational fraction R=A/B at a point x, as shown in the functional diagram below:

3:	A		3:	
2:	B	'VALF'	2:	
1:	x	——>	1:	A(x)/B(x)

N.B: 'VALF' calls program 'VALP' in the 'POLY' directory.

'VALF':(Checksum: # 39503d, Size: 46 bytes)

```
« ROT OVER VALP 3 ROLLD VALP / »
```

Example: (in one second)

3:	[1 3 2 1]			
2:	[-1 5 7]	'VALF'	2:	
1:	2	——>	1:	1.92307692308

The result thus obtained is 25/13.

INTEGER POWERS OF A RATIONAL FRACTION

'POWF' calculates the nth power (where n is a positive integer) of a rational fraction A/B. The functional diagram is as follows:

3:	A		3:	
2:	B	'POWF'	2:	A
1:	n	——>	1:	B

N.B: 'POWF' calls 'POWP' in the 'POLY' directory.

'POWF':(Checksum: # 8253d, Size: 43.5 bytes)

```
« ROT OVER POWP 3 ROLLD POWP »
```

Example: (in 4 seconds)

3:	[1 -2 3]			
2:	[2 1]	'POWF'	2:	[1 -6 21 -44 63 -54 27]
1:	3	——>	1:	[8 12 6 1]

DERIVATIVE OF A RATIONAL FRACTION

'DERVF' calculates the derivative C/D of a rational fraction A/B, where:

$$C = A'B-AB' \text{ and } D = B^2.$$

This gives the following functional diagram:

2:	A	'DERVF' ——>	2:	C
1:	B		1:	D

N.B: Program 'DERVF' of course calls program 'DERIV' in the 'POLY' directory. Programs 'PRODP', 'ADDP' and 'ELML' are also called.

'DERVF':(Checksum: # 56824d, Size: 121 bytes)

```
«  →  a  b
   «  a  DERIV  b  PRODP
      a  b  DERIV  PRODP  NEG
      ADDP  ELML
      b  DUP  PRODP
   »
»
```

Example: (in 3 seconds)

2:	[1 -2 3]	'DERVF' ——>	2:	[1 8 -11]
1:	[1 4]		1:	[1 8 16]

And we find that the derivative of:

$$R(X) = \frac{X^2 - 2X + 3}{X + 4} \quad \text{is} \quad R'(X) = \frac{X^2 + 8X - 11}{X^2 + 8X + 16}$$

DECOMPOSITION OF A RATIONAL FRACTION INTO PARTIAL FRACTIONS

'DECPF' allows you to decompose a rational fraction R=A/B, with real or complex coefficients, into partial fractions.

The numerator A must be given in vector form.

The denominator B must be given in list form. More specifically, the decomposition of B into products of irreducible factors is written:

$$B = B_1^{\alpha_1}\ B_2^{\alpha_2}\ \ldots.\ B_n^{\alpha_n} \text{ (where the } \alpha\text{'s are integers} \geq 1\text{),}$$

so B must be given in the following form:

$$B = \{\ B_1\ \ \alpha_1\ \ B_2\ \ \alpha_2\ \ \ldots.\ B_n\ \ \alpha_n\ \}.$$

All exponents α_i must be included in this list, even if equal to 1.
The various polynomials B_i must be relatively prime and irreducible (therefore first or second-degree but with real coefficients and a negative discriminant).

Program 'DECPF' does not make any check on the validity of the list given for B.
Should you fail to keep to the conditions described above, run errors will inevitably occur or results obtained will not be able to be interpreted properly.

The functional diagram is as follows:

```
                                (program suspended)
      |            |            |               |
2:    |     A      |            | decomposition |
      |            |  ------>   |               |
1:    |    List    |        1:  |  element(s)   |
```

The program progressively decomposes the fraction and gives each partial fraction, halting at each intermediate result.
The program can be resumed by pressing CONT (without having to leave the stack as it was) to obtain the next intermediate result.

Let us suppose, therefore, that the rational fraction R is written:

$$R = \frac{A}{B_1^{\alpha_1}\ B_2^{\alpha_2}\ \cdots\cdots\ B_n^{\alpha_n}}.$$ The results will be given in the following order:

First result: The integer part I (even if it is zero) in vector form (conventional polynomial notation in the 'POLY' directory).

Second result: The main part which gives the factor B_k as the denominator of the rational fraction R:

$$\frac{A_\alpha}{B_k^\alpha} + \frac{A_{\alpha-1}}{B_k^{\alpha-1}} + \frac{A_{\alpha-2}}{B_k^{\alpha-2}} + \cdots\cdots\cdots \frac{A_1}{B_k}$$

where A_α, $A_{\alpha-1}$, ..., A_1 are polynomials of a degree strictly less than the degree of B_k.
The result obtained here is a list written as follows:
List = { A_α, $A_{\alpha-1}$, ..., A_1 }, where the various polynomials A_i are written in vector form.

Subsequent results: We subsequently obtain a succession of results giving the other factors B_k as the denominator of the fraction R, as explained above.

Notes:

'DECPF' calls 'DIV' and 'ABCUV' in the 'POLY' directory.
'DECPF' has been made as short and as precise as possible. It can be improved upon by taking into account certain special points (especially if the denominator has at most two different factors, for example).
A special feature of 'DECPF' is that all the main parts are calculated with the same degree of accuracy. Results already obtained are not in fact used in decomposing the fraction. This would be possible (the intention being to save on computation time), but not without spreading round-off errors to a dangerous extent.

'DECPF':(Checksum: # 46365d, Size: 331.5 bytes)

```
«   DUP   SIZE   →   a   L1   n
    «   {}   [ 1 ]
        1   n   FOR   i
            L1   i   GETI   3   ROLLD   GET   POWP   ROT
            OVER   +   0   +   3   ROLLD   PRODP
        2   STEP
        →   L2   b
        «   a   b   DIV   'a'   STO   HALT
            1   n   FOR   i
                L2   i   GET   b   OVER   DIV   DROP   SWAP   a
                ABCUV   DROP   L1   i   GETI   3   ROLLD   GET
                →   d   k
                «   1   k   START   d   DIV   SWAP   NEXT
                    DROP   k   →LIST
                »
            HALT   2   STEP
»   »   »
```

PROGRAM 'DECPF': PRACTICAL EXAMPLE

We want to decompose the following rational fraction into partial fractions:

```
                  X^13
R = ─────────────────────────────────────
     (X-1)^2 * (X^2+X+1) * (X^2+1)^3
```

We therefore create the stack:

```
2: [ 1 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
1: { [1 -1] 2 [1 1 1] 1 [1 0 1] 3 }
```

and call program 'DECPF'.
The program halts after ten seconds and we find the vector [1 1 -2 -1] at level 1 of the stack. The integer part is therefore equal to `X^3+X^2-2X-1`.

Pursuing the program (with CONT), we obtain the following at level 1 of the stack (within 43 seconds):

```
1: { [ .041666666661 ]  [ .374999999959 ] }
```

(if we use →Q, we see that the numbers obtained are 1/24 and 3/8).
The main part corresponding to `(X-1)^2` is thus:

```
     1              3
────────── + ─────────
24*(X-1)^2     8*(X-1)
```

Continuing with CONT, we obtain after a further 25 seconds:

```
1: { [ .333333333344 ] }
```

The corresponding term of the decomposition is therefore:

```
      1
────────────
3*(X^2+X+1)
```

Continuing again with CONT, after 49 seconds we find:

```
   { [ .50000000001 .00000000004 ]
     [ -2.50000000003 -.25000000016 ]
1:   [ 4.62500000008 1.25000000036 ] }
```

We therefore obtain the next part of the decomposition:

```
     X              -10*X-1           37*X+10
──────────── + ──────────── + ───────────
2*(X^2+1)^3     4*(X^2+1)^2    8*(X^2+1)
```

We then press CONT again to terminate the program.
We therefore obtain a full decomposition into partial fractions of the initial rational fraction.
Notice that the results obtained are extremely good with low round-off errors.

MATRIX CALCULATIONS

The 'MATR' directory contains programs written for calculations performed on matrices or linear systems.
This is an area in which the HP48 really comes into its own, as it is capable of solving compute-intensive problems with relatively simple programs. The programmer no longer has to worry about the time-consuming calculation of products of matrices or vectors, as these are handled by the machine.
However, some of the programs shown here involve lengthy computation. Calculations on arrays in fact take quite a long time, for two main reasons:

- Firstly, we have to use indexed addressing to access a coefficient of a given matrix.
- Secondly, such arrays frequently need to be copied into the stack.

The 'MATR' directory contains the following programs:

'CB' : changes the matrix of a linear transformation (where the initial matrix and basis transformation matrix are known).

'TR' : calculates the trace of a square matrix.

'POWM' : calculates the powers of a square matrix.

'INVN' : gives the inverse of a square matrix A whose elements are integers, giving the determinant d and the matrix (whose coefficients are integers) `B=d*A^(-1)`.

'ALU' : lets you decompose a square matrix into the product of a lower triangular matrix with a unit diagonal and an upper triangular matrix.

'PUTR' : places a row in a matrix.

'PUTC' : places a column in a matrix.

'GETR' : extracts a row from a matrix.

'GETC' : extracts a column from a matrix.

'SWPR' : swaps two rows.

'SWPC' : swaps two columns.

'CALCR' : lets you perform calculations on the rows of a matrix.

'CALCC' : lets you perform calculations on the columns of a matrix.

'CRARY' : lets you create an array (matrix or vector) whose general term is given by a formula.

'CARP' : gives the characteristic polynomial of a square matrix.

'EV234' : gives the eigenvalues of a square matrix of order 2, 3 or 4.

'DEFL' : lets you approximate the eigenvalues and eigenvectors of matrices of an order greater than 4.

'RANK' : calculates the rank of a matrix (by Gaussian elimination).

'SYST' : gives the symbolic expression of the general solution of a system of n equations in p unknowns. Of particular interest when the system has an infinite number of solutions.

'EIGSP' : gives the equation(s) of the eigensubspace of a square matrix for a given eigenvalue.

'DIVAC' : allows you to divide an array by a square matrix more accurately than with '/'.

'INVAC' : allows you to invert a square matrix more accurately than with the INV command.

'ADDID' : bounds a matrix A (with n rows) to the right with the identity matrix of order n (useful when using programs 'MPOL', 'PIVOT' and 'EQLR').

'MPOL' : calculates the minimal polynomial of a square matrix.

'PIVOT' : employs the Gaussian elimination or pivot method applied to a matrix.

'EQLR' : finds any linear relations between n vectors or the equations of the vector space generated by the same n vectors.

CHANGING BASIS VIA A TRANSFORMATION MATRIX

'CB' calculates the new matrix B = P^{-1} A P of a linear transformation f, where the initial matrix is A and the transformation matrix used to change from the initial basis to the new basis is P.
The functional diagram is as follows:

```
2:  |     A     |     'CB'     2:  |           |
                 ------------>
1:  |     P     |              1:  |  P   A P  |
```

'CB': (Checksum: 31267# d, Size: 42.5 bytes)

```
« → p
   « p / p *
   »
»
```

Example:

```
    [[  3  -1   2  ]
     [  0   5   1  ]             'CB'    2:
2:   [  4  -2   7  ]]          -------->
    [[  1  -1   1  ]                        [[ -15  24  -13 ]
     [ -1   2   1  ]                         [ -13  21   -8 ]
1:   [  0   0   1  ]]                    1:  [   6  -8    9 ]]
```

CALCULATING THE TRACE OF A SQUARE MATRIX

'TR' calculates the trace tr(A) (i.e. the sum of the coefficients in the leading diagonal) of a square matrix A, as shown in the functional diagram below:

```
                      'TR'
1:  |     A     |    ----->   1:  |   tr(A)   |
```

'TR': (Checksum: # 16159d, Size: 68 bytes)

```
«  0
   1  3  PICK  SIZE  2  GET  FOR  i
      OVER  i  DUP  2  →LIST  GET  +
   NEXT
   SWAP  DROP
»
```

Example:

```
   [[ 3  5  -8 ]          'TR'        2:  |        |
    [ 7  1  11 ]       ----------->
1:  [ 4 -5   6 ]]                     1:  |   10   |
```

POWERS OF A SQUARE MATRIX
==============================

'POWM' calculates the powers A^n (where the exponent n is a positive or negative integer) of a square matrix A.
If the exponent is negative, the matrix A must be invertible.
If n is zero, the result is the identity matrix of the same order as A.

The functional diagram is as follows:

```
2:   A         'POWM'    2:
1:   n        ------->   1:   A
```

'POWM': (checksum: # 53967d, Size: 157 bytes)

```
«   IF   DUP   0   <   THEN
         SWAP  INV  SWAP  NEG
    END
    OVER  IDN
    →    n   p
    «    WHILE   n
         REPEAT
              IF   n   2   MOD   THEN
                   DUP        'p'   STO*
              END
              SQ   n   2   /   FLOOR   'n'   STO
         END
         DROP   p
    »
»
```

Example 1: (in under 2 seconds)

```
     [[  1   2   2  ]
      [  0  -1   0  ]
2:    [  1   1   3  ]]    'POWM'    [[ 110771   110770   302632 ]]
                         ------->    [    0        1        0    ]
1:        10                  1:    [ 151316   151316   413403 ]]
```

Example 2: (in 2 seconds)

```
     [[  1   2   2  ]
      [  0  -1   0  ]
2:    [  1   1   3  ]]    'POWM'    [[  571    572   -418  ]]
                         ------->    [    0     -1      0   ]
1:        -5                  1:    [ -209   -209    153  ]]
```

INVERSE OF A MATRIX WHOSE COEFFICIENTS ARE INTEGERS

'INVN' will allow you to precisely calculate the inverse of a square matrix A whose coefficients are integers.

The functional diagram is as follows:

```
    |            |                           |            |
    |------------|                           |------------|
 2: |            |         'INVN'         2: |     B      |
    |------------|      ------------>        |------------|
 1: |     A      |                        1: |     d      |
    --------------                           --------------
```

where "d" is the determinant of A (an integer) and B is the square matrix whose coefficients are integers such that `A^(-1)=(1/d)*B`.

'INVN': (Checksum: # 24340d, Size: 56.5 bytes)

```
«   DUP   DET   .5   +   FLOOR   SWAP
    INV   OVER  *    0   RND     SWAP
»
```

Example: we want to calculate the inverse of the matrix:

```
        [[ 1   3 -4 ]
  A =    [ -2 1   5 ]
         [ 7   3  2 ]]
```

Within two seconds, we find:

```
                                             |             |
    |              |                         |[[-13 -18 19]|
    |--------------|                         | [ 39  30  3]|
    |[[  1  3  -4 ]|       'INVN'         2: | [-13  18  7]]
    | [ -2  1   5 ]|    ------------>        |-------------|
 1: | [  7  3   2]]|                      1: |     156     |
    ----------------                         ---------------
```

The inverse of the matrix is therefore:

```
   1    [[ -13   -18   19 ]
 -----   [  39    30    3 ]
  156    [ -13    18    7 ]]
```

DECOMPOSITION "A=LU" OF A SQUARE MATRIX A

'ALU' decomposes a square matrix A into the product LU of a square matrix L (Lower triangular with unit diagonal) and a matrix U (Upper triangular).

N.B: Program 'ALU' will overwrite any variable 'L' in the directory. It calls programs 'GETR' and 'PUTR'.

The functional diagram is as follows:

2:		'ALU'	2:	U
1:	A	——————>	1:	L

'ALU': (Checksum: # 23599d, Size: 249.5 bytes)

```
«  DUP  SIZE  1  GET  DUP  IDN  'L'  STO
   →  d
   «  1  d  1  -  FOR  i
         DUP  i  GETR  DUP  i  GET
         →  Lp  p
         «  i  1  +  d  FOR  j
                  'L'  j  i  2  →LIST  3  PICK  OVER
                  GET  p  /  DUP  4  ROLLD  PUT  OVER
                  j  GETR  Lp  ROT  *  -  j  PUTR
            NEXT
         »
      NEXT
      L
      'L' PURGE
   »
»
```

Example: (in 8 seconds)

```
                                          [[  2  -2   3   1 ]
                                           [  0   3   1   5 ]
                                           [  0   0   1  -4 ]
2:                                     2:  [  0   0   0   7 ]]
                              ——————>
   [[  2   -2    3    1  ]     'ALU'      [[  1   0   0   0 ]
    [  6   -3   10    8  ]                 [  3   1   0   0 ]
    [  2  -14    0  -23  ]                 [  1  -4   1   0 ]
1:  [  6    0   18   -8  ]]            1:  [  3   2   7   1 ]]
```

PLACING A ROW IN A MATRIX

'PUTR' places a row R in a matrix A, at row number n, thus transforming matrix A into matrix B, as shown in the functional diagram below:

3	A		3	
2	R	'PUTR' ——>	2	
1	n		1	B

'PUTR': (Checksum: # 8768d, Size: 84 bytes)

```
«   {1}   +   SWAP
    →   L
    «   1   3   PICK   SIZE   2   GET   FOR   i
            L   i   GET   PUTI
        NEXT
        DROP
    »
»
```

N.B: R may be a row vector or a row matrix.

Example:

3:	[[4 7 5 1] [-3 10 -7 0] [5 -1 5 3] [-9 15 0 2]]			
2:	[7 1 8 9]	'PUTR' ——>		
1:	2		1:	[[4 7 5 1] [7 1 8 9] [5 -1 5 3] [-9 15 0 2]]

PLACING A COLUMN IN A MATRIX

'PUTC' places a column C in a matrix A, at column number n, thus transforming matrix A into matrix B.

<u>N.B</u>: 'PUTC' calls program 'PUTR'.
Column C may be written as a column vector or column matrix.

The functional diagram is as follows:

```
   |          |                     |          |
3  |    A     |                  3  |          |
2  |    C     |     'PUTC'       2  |          |
1  |    n     |    -------->     1  |    B     |
```

'PUTC': (Checksum: # 59798d, Size: 53.5 bytes)

```
«   ROT   TRN   ROT
    DUP   SIZE  1   1   SUB   RDM
    ROT   PUTR  TRN
»
```

<u>Example</u>:

```
    |                    |
    | [[  4   7  5  1  ] |
    |  [ -3  10 -7  0  ] |
    |  [  5  -1  5  3  ] |
3:  |  [ -9  15  0  2  ]]|               |                     |
2:  |  [  7   1  8  9  ] |    'PUTC'     | [[  4   7  7  1  ]  |
1:  |         3          |   -------->   |  [ -3  10  1  0  ]  |
                                         |  [  5  -1  8  3  ]  |
                                     1:  |  [ -9  15  9  2  ]] |
```

EXTRACTING A ROW FROM A MATRIX

'GETR' extracts row number n from a matrix A, the result being a vector R.

The functional diagram is as follows:

```
                      'GETR'
2:      A          ---------->   2:
1:      n                        1:      R
```

'GETR': (Checksum: # 60021d, Size: 51 bytes)

```
«   SWAP   TRN   DUP   SIZE   2   2   SUB
    0   CON   ROT   1   PUT   *
»
```

Example:

```
    [[ 1  -8   5   7   3 ]
     [ 4   5  11  -9   4 ]
2:   [ 3  -1   0   3   1 ]]        'GETR'
1:             3                  ------->   1:  [ 3  -1  0  3  1 ]
```

EXTRACTING A COLUMN FROM A MATRIX

'GETC' extracts column number n from a matrix A, the result being a column matrix C.

The functional diagram is as follows:

```
2:  |    A    |    'GETC'    2:  |         |
              |   ------->       |         |
1:  |    n    |              1:  |    C    |
```

'GETC': (Checksum: # 41771d, Size: 48.5 bytes)

```
«   OVER   SIZE   1   1   PUT   0   CON
    TRN    SWAP   1   PUT   *
»
```

Example:

```
    | [[  1  -8   5   7   3 ] |
    |  [  4   5  11  -9   4 ] |
2:  |  [  3  -1   0   3   1 ]]|    'GETC'
                                  ------->
1:  |            4            |         1:  |   [[  7  ]    |
                                            |    [ -9  ]    |
                                            |    [  3  ]]   |
```

SWAPPING TWO COLUMNS OF A MATRIX

'SWPC' swaps the columns numbered n and m of a matrix A, thus transforming matrix A into a matrix B.

The functional diagram is as follows:

```
3:  |  A  |
2:  |  m  |      'SWPC'
1:  |  n  |    --------->   1:  |  B  |
```

'SWPC': (Checksum: # 47058d, Size: 126 bytes)

```
«  →   m   n
   «   DUP   SIZE   2   GET   IDN   m   m   2   →LIST   0   PUT
       n   n   2   →LIST   0   PUT   m   n   2   →LIST   1   PUT
       n   m   2   →LIST   1   PUT   *
   »
»
```

<u>Example</u>:

```
[[  1  -9   6  ]
 [  4   3  -9  ]
 [ 11   5   7  ]]
        2            'SWPC'        [[  1   6  -9  ]
                   --------->       [  4  -9   3  ]
        3                      1:   [ 11   7   5  ]]
```

SWAPPING TWO ROWS OF A MATRIX

'SWPR' swaps the rows numbered n and m of a matrix A, thus transforming matrix A into a matrix B.

The functional diagram is as above for 'SWPC'. 'SWPR' also calls 'SWPC'.

'SWPR': (Checksum: # 19714d, Size: 38.5 bytes)

```
«   ROT   TRN   3   ROLLD   SWPC   TRN   »
```

<u>Example</u>:

```
[[  1  -9   6  ]
 [  4   3  -9  ]
 [ 11   5   7  ]]
        1            'SWPR'        [[ 11   5   7  ]
                   --------->       [  4   3  -9  ]
        3                      1:   [  1  -9   6  ]]
```

CALCULATIONS ON THE ROWS OF A MATRIX

'CALCR' allows you to perform calculations on the rows of a matrix A, and to modify rows if required. There are two types of functional diagram:

```
2:  |     A      |    'CALCR'    2:  |     A      |
1:  | Expression |  ---------->  1:  |   Result   |
```

```
3:  |     A      |               3:  |            |
2:  | Expression |    'CALCR'    2:  |            |
1:  |     n      |  ---------->  1:  |     B      |
```

In both cases, "Expression" is an algebraic expression or a program (written in RPN notation) in which the rows of A are denoted by the names 'R1', 'R2', 'R3', etc.
In the first case, "Result" indicates the result of the evaluation of "Expression". This may be a vector (in the same format as the rows of A) or any other object (particularly if "Expression" is in fact an RPN program). If "Expression" is an RPN program and does not behave on the stack like an algebraic expression, the stack may not look the way it does here after computation.
In the second case, n denotes the number of the row of A to be modified. 'CALCR' places the result of the evaluation of "Expression" in the nth row of A (the result must of course be a vector in the same format as the rows of A) and 'B' denotes the matrix thus modified.

N.B: 'CALCR' creates the global variables 'R1', 'R2', etc. (for as many rows as there are in the matrix) before purging them. It also calls programs 'GETR' and 'PUTR'.

'CALCR': (Checksum: # 40802d, Size: 218.5 bytes)

```
«   DUP  TYPE   NOT   DUP   DROPN   3   PICK   SIZE   1   GET
    «    "'R"   SWAP   →STR   +   OBJ→   »   RCLF
    →    n  t  p  f
    «    STD
         1  t   FOR  i
              OVER  i  GETR  i  p  EVAL  STO
         NEXT
         EVAL
         1  t   FOR  i  i  p  EVAL  PURGE  NEXT
         IF  n  THEN  n  PUTR  END
         f  STOF
    »
»
```

PROGRAM 'CALCR': PRACTICAL EXAMPLES

Example 1: in 4 seconds,

```
    [[ 1  -1  3  0  ]                    [[ 1  -1  3  0  ]
     [ 0   5  2  4  ]                     [ 0   5  2  4  ]
2:   [ 6  -2  8 -7  ]]               2:   [ 6  -2  8  -7 ]]
                            'CALCR'
1:   'R1+2*R3-R2'           ------>  1:  [ 13 -10  17 -18 ]
```

This example simply evaluates the expression:

```
'R1+2*R3-R2', with R1 = [ 1  -1  3  0 ]
                   R2 = [ 0   5  2  4 ]
               and R3 = [ 6  -2  8 -7 ],
```

and enters the result at level 1.

Example 2: in five seconds,

```
    [[ 1  -1  3  0  ]
     [ 0   5  2  4  ]
3:   [ 6  -2  8 -7  ]]

2:   'R1+2*R3-R2'                        [[ 13 -10 17 -18 ]
                            'CALCR'       [  0   5  2   4 ]
1:                 1        ------>  1:   [  6  -2  8  -7 ]]
```

Here, we take the data previously obtained and tell 'CALCR' to place the result in row 1 of the matrix.

Example 3: in four seconds,

```
                                         [[ 1  -1  3  0  ]
    [[ 1  -1  3  0  ]                     [ 0   5  2  4  ]
     [ 0   5  2  4  ]                4:   [ 6  -2  8  -7 ]]
2:   [ 6  -2  8 -7  ]]               3:        1
    «  R1  R2  DOT                   2:        32
       R1  R3  DOT          'CALCR'
1:     R2  R3  DOT  »       ------>  1:       -22
```

In this example, we evaluate a program that successively calculates the scalar products of R1 and R2, R1 and R3, then R2 and R3.
We can see here that it is impossible to use an algebraic expression (as the instruction DOT cannot be used).
Furthermore, we can also see that the contents of the stack once 'CALCR' has been run may depend on what you put into the program to be evaluated.

CALCULATIONS ON THE COLUMNS OF A MATRIX

'CALCC' allows you to perform calculations on the columns of a matrix A, and to modify columns if required. There are two types of functional diagram:

```
2: |     A      |    'CALCC'    2: |    A     |
1: | Expression |  ----------->  1: |  Result  |

3: |     A      |               3: |          |
2: | Expression |    'CALCC'    2: |          |
1: |     n      |  ----------->  1: |    B     |
```

In both cases, "Expression" is an algebraic expression or a program (written in RPN notation) in which the columns of A are denoted by the names 'C1', 'C2', 'C3', etc.
In the first case, "Result" indicates the result of the evaluation of "Expression". This may be a column matrix (in the same format as the columns of A) or any other object (particularly if "Expression" is in fact an RPN program). If "Expression" is an RPN program and does not behave on the stack like an algebraic expression, the stack may not look the way it does here after computation.
In the second case, n denotes the number of the column of A to be modified. 'CALCC' places the result of the evaluation of "Expression" (via program 'PUTC') in the nth column of A (the result must of course be a vector or a column matrix) and 'B' denotes the matrix thus modified.

N.B: 'CALCC creates the global variables 'C1', 'C2', etc. (for as many columns as there are in the matrix) before purging them. It also calls programs 'GETC' and 'PUTC'.

'CALCC': (Checksum: # 39066d, Size: 218.5 bytes)

```
«  DUP   TYPE   NOT   DUP   DROPN   3   PICK   SIZE   2   GET
   «  "'C"   SWAP   →STR   +   OBJ→   »   RCLF
   →  n   t   p   f
   «  STD
      1   t   FOR   i
          OVER   i   GETC   i   p   EVAL   STO
      NEXT
      EVAL
      1   t   FOR   i   i   p   EVAL   PURGE   NEXT
      IF   n   THEN   n   PUTC   END
      f   STOF
   »
»
```

PROGRAM 'CALCC': PRACTICAL EXAMPLES

Example 1: in five seconds,

```
    [[  1  -1   3  0   ]              2: [[  1   -1   3   0  ]
     [  0   2   2  4   ]                  [  0    2   2   4  ]
 2:  [  6  -2   8 -7   ]]                 [  6   -2   8  -7  ]]
                            'CALCC'           [[  3   ]
 1:  'ABS(C2)*(C4+C1)'     -------->   1:      [ 12   ]
                                               [ -3   ]]
```

This example simply evaluates the expression:
`'ABS(C2)*(C4+C1)'`, with:

```
       [[ 1 ]          [[ -1 ]          [[ 0 ]
  C1 =  [ 0 ]     C2 =  [  2 ]     C4 =  [ 4 ]
        [ 6 ]]          [ -2 ]]          [-7 ]]
```

(therefore `ABS(C2)=√(1+4+4)=3`) and enters the result at level 1.

Example 2: in five to six seconds,

```
    [[  1  -1   3  0   ]
     [  0   2   2  4   ]
 3:  [  6  -2   8 -7   ]]
 2:  'ABS(C2)*(C4+C1)'                 [[  1 -1   3   0 ]
                            'CALCC'     [  0  2  12   4 ]
 1:          3             -------->  1: [  6 -2  -3  -7 ]]
```

Here, we take the data previously obtained and tell 'CALCC' to place the result in column 3 of the matrix.

Example 3: in five seconds,

```
                                       [[  1   -1   3   0  ]
    [[  1  -1   3  0   ]                [  0    5   2   4  ]
     [  0   2   2  4   ]            3:  [  6   -2   8  -7  ]]
 2:  [  6  -2   8 -7   ]]           2:      [[ -42 ]]
     «  C1   TRN   C4   *   'CALCC'
 1:     C3   {3}   RDM  »  -------->  1:    [ 3 2 8 ]
```

In this example, we evaluate a program that calculates:

* The product of the transpose of column C1 and column C4 to obtain the matrix [[-42]].
* The vector obtained by redimensioning column C3, which is equal to [3 2 8].

Here, the use of the commands **TRN** and **RDM** means that a program and not an expression has to be evaluated.

CREATING AN ARRAY WHOSE GENERAL TERM IS GIVEN BY A FORMULA

'CRARY' creates an array (a vector or matrix array) a whose general term (A(I) for a vector, A(I,J) for a matrix) is given by the formula F(I) (for a vector) or F(I,J) (for a matrix).

The size of the array to be created must be entered at level 1.

The calculation formula must be entered at level 2. This formula is an algebraic expression or a program giving the value of each element of the array for a given value of I (row number) and J (column number).

The functional diagram for the creation of a vector is as follows:

```
    |            |                       |            |
    |------------|                       |------------|
 2: |    F(I)    |     'CRARY'        2: |            |
    |------------|   ----------->        |------------|
 1: |   { m }    |                    1: |     A      |
    +------------+                       +------------+
```

The functional diagram for the creation of a matrix is as follows:

```
    |            |                       |            |
    |------------|                       |------------|
 2: |   F(I,J)   |     'CRARY'        2: |            |
    |------------|   ----------->        |------------|
 1: |  { m,n }   |                    1: |     A      |
    +------------+                       +------------+
```

'CRARY': (Checksum: # 3883d, Size: 147.5 bytes)

```
«   "« → I J"   ROT   →STR   +   OBJ→
    →  d  f
    «  d   OBJ→   DUP   1   ≠   DROPN
       →   L   c
       «   1   L   FOR   i
               1   c   FOR   j
                   i   j   f   EVAL
               NEXT
           NEXT
           d   →ARRY
       »
    »
»
```

PRACTICAL EXAMPLES USING PROGRAM 'CRARY'

Example 1: To create a vector of length 4 with a general term A(I)=I^3 (computation time under two seconds).

```
2: | 'I^3'  |    'CRARY'      |
1: | { 4 }  |   -------->  1: | [ 1  8  27  64 ] |
```

Example 2: To create a 3-row, 4-column matrix with a general term A(I,J) equal to I when I=J and J^2 when I ≠ J.

```
2: | '(I==J)*I + (I ≠ J)*J^2' |  'CRARY'     | [[ 1  4  9  16 ]
1: |        { 3 4 }           | -------->  1:|     1  2  9  16 ]
   |                          |              |     1  4  3  16 ]]
```

Example 3: To create a square matrix of order 5 with a general term A(I,J)=GCD(I,J). In this example, the general term is calculated using a program that goes into the 'ARIT' directory, calculates the GCD of I and J, then comes back into the 'MATR' directory (computation time approximately 4 seconds).

```
                                               | [[ 1  1  1  1  1 ]
                                               |  [ 1  2  1  2  1 ]
2: | « I J ARIT GCD MATR » |   'CRARY'         |  [ 1  1  3  1  1 ]
1: |        { 5 5 }        |  -------->   1:   |  [ 1  2  1  4  1 ]
                                               |  [ 1  1  1  1  5 ]]
```

CALCULATING THE CHARACTERISTIC POLYNOMIAL OF A SQUARE MATRIX

'CARP' calculates the characteristic polynomial P(x)=det(xI-A) of a square matrix A.
If A is a square matrix of order n, then P(x) is a nth- degree polynomial:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_k x^k + \ldots + a_1 x + a_0$$

Where:

$a_n = 1$, $a_{n-1} = -tr(A)$ (tr(A)=trace of A), $a_0 = (-1)^n \det(A)$.

The roots of the characteristic polynomial P of A are also the eigenvalues of A.

The polynomial P is given in vector form:

$[\ a_n \quad a_{n-1} \quad \ldots \quad a_1 \quad a_0\ ]$.

N.B: Program 'CARP' calls program 'TR' (to calculate the trace of a square matrix).

The functional diagram is as follows:

```
1: [    A    ]   --'CARP'-->   1: [    P    ]
```

'CARP': (Checksum: # 20812d, Size: 137 bytes)

```
«  DUP  SIZE  2  GET  DUP  IDN
   →  a  n  id
   «  1  id  1  n  FOR  k
            a  *  DUP  TR  k  /  NEG
            DUP  3  ROLLD  id  *  +
      NEXT
      DROP  n  1  +  →ARRY
   »
»
```

Example: (calculation time 4 seconds)

```
   [[  1   9  -5   2  ]
    [  4   0   6   7  ]
    [  3  -4  -1   2  ]     'CARP'
1:  [  5  -3   1   1  ]]   -------->  1: [ 1  -1  11  -495  -1336 ]
```

EIGENVALUES OF A SQUARE MATRIX OF ORDER 2, 3 OR 4 WITH REAL OR COMPLEX COEFFICIENTS

'EV234' calculates the eigenvalues of a square matrix A of order 2, 3 or 4 and with real or complex coefficients.
The eigenvalues of A are the coefficients k such that the system A(X)=kX has non-zero solutions. They are also the roots of the characteristic polynomial of A.
'EV234' works as follows:

- it first calculates the characteristic polynomial (program 'CARP');
- it then goes into the 'POLY' directory;
- it calculates the roots of the characteristic polynomial (using 'DEG2', 'DEG3' and 'DEG4' in the 'POLY' directory);
- then it returns to the 'MATR' directory.

The eigenvalues are placed in a list at level 1 of the stack. The number of times a multiple eigenvalue appears corresponds to its multiplicity.
Calculation times are approximately:
2 seconds for a 2x2 square matrix.
4 seconds for a 3x3 square matrix.
10 seconds for a 4x4 square matrix.

The functional diagram is as follows:

```
          ┌──────────────┐      'EV234'         ┌──────────────┐
     1:   │      A       │  ─────────────>  1:  │    liste     │
          └──────────────┘                      └──────────────┘
```

'EV234': (Checksum: # 14042d, Size: 130.5 bytes)

```
«   DUP    SIZE    2    GET
    →   n
    «   IF    { 2 3 4 }    n      POS     THEN
              CARP      POLY    "DEG"     n    →STR    +
              1    4    SUB     OBJ→      MATR
        END
        n     →LIST
    »
»
```

Example: (Result in Mode 4 FIX)

```
     ┌─────────────────────────┐                  ┌──────────────────────────┐
     │[[  3   -1   1   -1  ]   │    'EV234'       │{  (2.0000,0.0000)        │
     │ [  1    3   1    1  ]   │  ──────────>     │   (2.0000,0.0000)        │
     │ [ -1    1   1    1  ]   │                  │   (4.0000,0.0000)        │
 1:  │ [ -1    1  -3    5  ]]  │              1:  │   (4.0000,0.0000)  }     │
     └─────────────────────────┘                  └──────────────────────────┘
```

Matrix A above therefore has 4 real eigenvalues:
2 is a double eigenvalue.
4 is a double eigenvalue.

EIGENVALUES AND EIGENVECTORS OF A REAL SQUARE MATRIX (DEFLATION METHOD)

===

'DEFL' lets you calculate the approximate eigenvalues and associated eigenvectors of a square matrix A, of order n, with real coefficients, where A has n real and separate eigenvalues
$k_1, k_2, \ldots, k_n$ such that $|k_1| < |k_2| < \ldots < |k_n|$.

Functional diagram:

				- - - - - - - - - -
			5:	eigenvector u_3
4:			4:	eigenvalue k_2
3:			3:	eigenvector u_2
2:		'DEFL'	2:	eigenvalue k_1
1:	A	————>	1:	eigenvector u_1

'DEFL' halts whenever an eigenvalue k is obtained and the value is displayed at level 2 of the stack. The unit eigenvector for the eigenvalue k appears at level 1. We then find all the values up to the last eigenvalue by pressing CONT.

'DEFL' uses an iterative algorithm for finding eiegenvalues. Iteration halts when a value 'eps', which is equal by default to 1E-8, is reached (see this value in the text of the program). This value can be modified by inserting a new value for 'eps' at level 1 of the stack (before calling 'DEFL'), in which case the matrix A is shifted up to level 2. The smaller 'eps' is the better the approximation is, but calculation times are increased.

N.B: eigenvalues are calculated in order of decreasing absolute value. The speed with which an eigenvalue is calculated will increase proportionally with the size of its absolute value, with respect to other eigenvalues with lesser absolute values.

Example:

Let matrix A be equal to:

```
      [[   1   -9   -9   -9  ]
  A =  [ -13   19    0   22  ]
       [ -13   22   -3   22  ]
       [  26  -31   13  -34  ]]
```

We find (in 8 FIX mode):

In 12 seconds: `k =-24.9999989` and
`    u = [ -2.0116359E-9   0.40824829   0.40824829     -.81649658  ].`
In 24 seconds: `k =10.00000001` and
`    u = [ -0.50000000     0.50000000   0.50000000     -0.50000000 ].`
In 23 seconds: `k =-3.00000000` and
`    u = [ 3.97471899E-9   -0.70710678   -3.58306366E-9   0.70710678 ].`
In 15 seconds: `k = 1.00000000` and
`    u = [ 0.63245553   -.316222777   -.316222777     0.63245553 ].`

We can therefore say that the eigenvalues of A are -25, 10, -3 and 1, and that the corresponding eigenvectors associated with these values are:

`[0,1,1,-2]`, `[-1,1,1,-1]`, `[0,1,0,-1]` and `[2,-1,-1,2]` respectively.

TEXT OF PROGRAM 'DEFL'

'DEFL': (Checksum: # 29049d, Size: 485 bytes)

```
«   IF    DUP    TYPE    THEN    .00000001    END
    SWAP    DUP    SIZE    2    2    SUB    DUP    0    CON
    DUP    ROT    1    GET

    →   eps    a    old    new    p
    «
        «   →   mat
            «   old   1
                1   p   START   RAND   PUTI   NEXT   DROP
                DO
                    'old'   STO   mat   old   *   DUP   ABS   /
                    IF   DUP   old   DOT   0   <   THEN   NEG   END
                    DUP   'new'   STO
                UNTIL   'ABS   (new-old)   <   eps'   END
                mat   new   *   DOT   new   ABS   /   new
            »
        »
        →   evmax
        «   1   p   FOR   k
                a   evmax   EVAL
                IF   k   p   <   THEN
                    HALT
                    DUP2   a   TRN   evmax   EVAL
                    ROT   DUP2   DOT   /
                    DUP   SIZE   1   +   RDM   SWAP
                    1   OVER   SIZE   +   RDM   *
                    3   ROLLD   +   2   /   *
                    'a'   SWAP   STO-
                END
            NEXT
        »
    »
»
```

N.B: in the practical example given above, the results depend on the HP48's random number generator and might therefore differ from those I myself have obtained.

CALCULATING THE RANK OF A MATRIX

'RANK' calculates the rank of a matrix A (i.e. the number of linear independent columns or rows in A), having first transformed A into an upper triangular matrix B (using the Gaussian pivot method).

Functional diagram:

2:		'RANK'	2:	B
1:	A	——————>	1:	RANK(A)

N.B: program 'RANK' calls program 'SWPR'.

'RANK' rounds off to 7 decimal places to avoid mistakes due to round-off errors and to ensure that the pivot is of sufficient magnitude.
The final form of the matrix is also rounded off to 9 decimal places, which theoretically prevents any zero coefficients from appearing.
The instructions "7 RND" and "9 RND" can be modified if you feel that figures are being rounded off too strictly or not strictly enough.

'RANK': (Checksum: # 49944d, Size: 582 bytes)

```
«   DUP    TYPE    NOT    DUP    DROPN    OVER    SIZE    EVAL
    1   1   0   0   →   f   nr   nc   i   j   rk   rp
    «   «   0   i   nr   FOR   ii
                OVER   ii   j   2   →LIST   GET   ABS   DUP2
                IF   <   THEN   ii   'rp'   STO   SWAP   END
            DROP   NEXT
        »
        «   DUP   i   j   2   →LIST   GET   →   piv
            «   nr   IDN
                1   f   NOT   i   *   +   nr   FOR   ii
                    IF   ii   i   ≠   THEN
                        ii   i   2   →LIST   3   PICK
                        ii   j   2   →LIST   GET   piv
                        /   NEG   PUT
                    END
                NEXT   SWAP   *
            »
        »   →   fp   pv
        «   WHILE   i   nr   ≤   j   nc   ≤   AND
            REPEAT   fp   EVAL
                IF   7   RND   THEN
                    'rk'   1   STO+
                    IF   i   rp   <   THEN   i   rp   SWPR   END
                    IF   i   nr   <   f   OR   THEN   pv   EVAL   END
                    'i'   1   STO+
                END
                'j'   1   STO+
            END
            9   RND   rk
»   »   »
```

PRACTICAL EXAMPLE USING PROGRAM 'RANK'

Example 1: (in approximately 5 seconds)

```
      |                                  |                  |                                  |
      |                                  |                  | [[ 5     -9      6       2  ]    |
      |----------------------------------|   'RANK'  2:     |    0   -1.6    -.6    -2.2  ]    |
      | [[   2    -5    3    1      ]    |   ------->       |    0     0   -2.25  -2.25 ]      |
      |  [   3    -7    3   -1      ]    |                  |    0     0      0      1  ]]    |
      |  [   5    -9    6    2      ]    |                  |----------------------------------|
  1:  |  [   4    -6    3    1      ]]   |            1:    |               4                  |
```

Example 2: (in approximately 13 seconds)

Matrix A equal to:

```
         [[  18    -12    10    11    -9    -19    10   ]
          [  4      5      9     5     2     1      3   ]
  A =     [  1      8      5     2     4     6      1   ]
          [  10     5      3     4     1     2      5   ]
          [  -6     0      6     1     1    -1     -2   ]]
```

is a matrix of rank 3, and the triangular matrix B obtained is written (in 2 FIX mode):

```
[[    18      -12      10      11      -9     -19      10  ]
 [    0      11.67   -2.56   -2.11     6     12.56    -.56 ]
 [    0        0      8.46    3.94    .06    -3.03    1.14 ]
 [    0        0       0       0       0       0        0  ]
 [    0        0       0       0       0       0        0  ]]
```

Variant:

If we put a non-zero real number at level 1 of the stack before calling 'RANK' (matrix A is shifted up to level 2), triangulation of the resulting matrix B obtained is taken further, as all the coefficients above a non-zero pivot have been cancelled out.
This variant is used in programs 'SYST' (symbolic resolution of a linear system) and 'EIGSP' (equations of eigensubspaces). In the example above, the matrix obtained using this variant (still in 2 FIX mode) would be:

```
[[    18       0       0     5.39   -2.88   -3.45    8.43 ]
 [    0      11.67     0     -.92    6.02   11.64    -.21 ]
 [    0        0     8.46    3.94    .06    -3.03    1.14 ]
 [    0        0       0       0       0       0       0  ]
 [    0        0       0       0       0       0       0  ]]
```

SYMBOLIC SOLUTION OF A SYSTEM OF LINEAR EQUATIONS

'SYST' lets you find the symbolic solution of a system of n linear equations in p unknowns:

$$\begin{cases} a_{11}\ x_1 + a_{12}\ x_2 + \ldots\ldots + a_{1p}\ x_p = b_1 \\ a_{21}\ x_1 + a_{22}\ x_2 + \ldots\ldots + a_{2p}\ x_p = b_2 \\ \vdots \\ a_{n1}\ x_1 + a_{n2}\ x_2 + \ldots\ldots + a_{np}\ x_p = b_n \end{cases}$$

```
2: |    A    |     'SYST'           |  Symbolic     |
1: |    B    |  ----------->    1:  |  expression   |
                                      |  of solution. |
```

If we write the matrix of the system A and the vector of second members B, we get the following functional diagram:

The symbolic expression of the solution is:

- The message "No solution" if the system has no solution.
- A list containing the equation(s) defining the general solution of the system. These may be in the following forms:

`'X3=-17'` (for example) if -17 is the only value of the unknown X3 that satisfies the system; or:
`'X2=3*X4-2*X5'` (for example) if the solution of the system expresses the unknown X2 in terms of the unknowns X4 and X5 (which are both undetermined).
(The unknowns of the system are written X1, X2, X3, ...).

N.B: program 'SYST' calls program 'RANK'.

Example: (in 16 seconds)

Solve the system:

$$\begin{cases} 2X_1 + 5X_2 + 10X_3 + 17X_4 + 26X_5 = 1 \\ 3X_1 + 6X_2 + 11X_3 + 18X_4 + 27X_5 = 2 \\ 4X_1 + 7X_2 + 12X_3 + 19X_4 + 28X_5 = 3 \end{cases}$$

(Result in 8 FIX mode)

```
   [[ 2  5  10  17  26 ]                  { 'X1=1.33333333
    [ 3  6  11  18  27 ]     'SYST'         +1.66666667*X3
2:  [ 4  7  12  19  28 ]]    ------>        +4*X4+7*X5'
                                            'X2=-0.33333333
1:      [ 1  2  3 ]                  1:     -2.66666667*X3
                                            -5*X4-8*X5 }
```

The above system therefore has an infinite number of solutions. The values of X3, X4 and X5 are arbitrary and X1 and X2 are given respectively by:

$$X_1 = 4/3 + 5X_3\ /3 + 4X_4 + 7X_5$$
$$X_2 = -1/3 - 8X_3\ /3 - 5X_4 - 8X_5$$

TEXT OF PROGRAM 'SYST'

'SYST': (Checksum: # 8818d, Size: 464 bytes)

```
«  →STR   SWAP  TRN   →STR   1   OVER   SIZE   1   -   SUB
   SWAP   +   OBJ→   TRN   1   RANK   OVER   SIZE   2   GET
   «   RCLF   STD   "'X"   ROT   →STR   +   OBJ→   SWAP   STOF   »

         →   a   rk   nc   var

         «   { }
             1   rk   FOR   i

                 a   i   { 1 }   +   0
                 DO   DROP   GETI   UNTIL   DUP   EVAL   END
                 SWAP   2   GET
                 IF   DUP   1   ==   THEN
                      3   DROPN   "No solution"   DOERR
                 END
                 1   -

             →   piv   j

                 «   DROP   j   var   EVAL   'a(i,nc)/piv'   EVAL
                     IF   j   1   +   nc   <   THEN
                          j   1   +   nc   1   -   FOR   jj
                              '-a(i,jj)/piv'   EVAL
                              jj   var   EVAL   *   +
                          NEXT
                     END
                 »

                 =   +

             NEXT

         »
   »
```

EQUATIONS OF EIGENSUBSPACES OF A SQUARE MATRIX

'EIGSP' lets you find the equation(s) of the eigensubspace of a square matrix A, with respect to an eigenvalue k. It therefore calls program 'SYST' to solve the system `(A-kI)X=0` symbolically, where I is the identity matrix of the same order as A and X is a vector whose components are denoted by X1, X2, X3, etc.

Functional diagram:

```
     |            |                    |              |
     |------------|                    |--------------|
  2: |     A      |      'EIGSP'       | Equations of |
     |------------|   ------------->   |   an eigen-  |
  1: |     k      |                 1: |   subspace   |
     |------------|                    |--------------|
```

The equation(s) is/are given in list form and according to the syntax of the program 'SYST'. If k is not a true eigenvalue of A, the result is `{ 'X1=0' 'X2=0' 'X3=0' ...}`

'EIGSP': (Checksum: # 44125d, Size: 54.5 bytes)

```
«   OVER   IDN   *   -   DUP   SIZE
    1   1   SUB   0   CON   SYST
»
```

Example: (in 12 seconds)

```
   |                              |
   |[[  3   1   1   1   ]         |  'EIGSP'
   | [  1   3   1   1   ]         |  ------>   |------------------------------|
2: | [  1   1   3   1   ]         |         1: |{'X1=X4' 'X2=X4' 'X3=X4'}     |
   | [  1   1   1   3   ]]        |            |------------------------------|
   |------------------------------|
1: |            6                 |
   |------------------------------|
```

In other words, 6 is a single eigenvalue of the matrix and the eigensubspace is the straight line corresponding to the vector generated by [1 1 1 1] (obtained by giving a value of 1 to the variable X4).

Similarly, with the same matrix and with k=2, the result is:

`{ 'X1=-X2-X3-X4' }`

meaning that 2 is a triple eigenvalue, the eigensubspace being generated by the vectors: `[ -1 0 0 1 ]`, `[ -1 0 1 0 ]` and `[ -1 1 0 0 ]` (obtained by giving a value of 1 to one of the variables X2, X3 and X4 and a value of 0 to the other two).

DIVIDING MATRICES WITH IMPROVED ACCURACY

'DIVAC' allows you to obtain greater accuracy than with the / command (dividing an array by a matrix) by using the RSD instruction on the HP48. This gives us the following functional diagram:

```
2: |     A      |    'DIVAC'    2: |            |
                  ------------>
1: |     B      |               1: | B^(-1)*A   |
```

where A is an array and B is a square matrix.

'DIVAC': (Checksum: # 63411d, Size: 44.5 bytes)

« DUP2 / 3 DUPN RSD ROT / + SWAP DROP »

Example:

```
           [[ 38 61 ]          [[ 5 -3  7 ]
With A =  [ 87 67 ]  and B =  [ 8  1  6 ]
           [-31 -7 ]]          [ 4 -5 -2 ]]
```

we find:

Using /:

```
              [[ 5.99999999998  2.99999999999 ]
B^(-1)*A =  [ 9.00000000002  .999999999994 ]
              [ 5.00000000003  7.00000000001 ]]
```

and using 'DIVAC':

```
              [[ 6  3 ]
B^(-1)*A =  [ 9  1 ]
              [ 5  7 ]]
```

which is the exact result.

INVERSE OF MATRICES WITH IMPROVED ACCURACY

'INVAC' lets you find the inverse of a square matrix with greater accuracy than with the INV command (which has the same function as the 1/x key on the keypad).
The functional diagram is as follows ('INVAC' calls 'DIVAC'):

```
                      'INVAC'
1: |     A     |   ------------>   1: |  A^(-1)  |
```

'INVAC': (Checksum: # 64408d, Size: 35.5 bytes)

« DUP IDN SWAP <u>DIVAC</u> »

Examples:

```
           [[-2 -3  2 ]
With A =  [ 6  1  6 ] ,
           [ 4  5 -2 ]]
```

we find

Using INV:

```
            [[ -4.00000000022  .500000000027 -2.50000000014 ]
A^(-1)=  [  4.50000000026 -.500000000031  3.00000000016 ]
            [  3.25000000018 -.250000000021  2.00000000011 ]]
```

and using 'INVAC':

```
            [[ -4     .5   -2.5 ]
A^(-1)=  [  4.5  -.5    3   ]
            [  3.25 -.25   2   ]]
```

which is the exact result.

BOUNDING A MATRIX TO THE RIGHT WITH THE IDENTITY MATRIX

'ADDID' lets you bound a matrix A in the format {n,p} to the right with the identity matrix of order n. The result is a matrix B in the format { n, n+p }.

'ADDID' is called when running program 'EQLR'. It is also most useful when inverting a matrix using the Gaussian pivot method (see program 'PIVOT').

The functional diagram is as follows:

```
       ┌──────────┐   'ADDID'          ┌──────────┐
    1: │    A     │  ──────────>   1:  │    B     │
       └──────────┘                    └──────────┘
```

'ADDID': (Checksum: # 38199d, Size: 135.5 bytes)

```
«   TRN     DUP     SIZE    EVAL
    →   p   n
    «   p   n   +   n   2   →LIST   RDM
        p   n   *   1   +   n   p   +   n   *   FOR
i
            i   1   PUT
        n   1   +   STEP
        TRN
    »
»
```

Example: (in one second)

```
    [ [ 1  3  2 ]                  [[ 1  3  2  1  0  0  0  0  0 ]
      [ 8  7  5 ]      'ADDID'      [ 8  7  5  0  1  0  0  0  0 ]
      [ 6  4  9 ]    ──────────>    [ 6  4  9  0  0  1  0  0  0 ]
      [ 0  3  8 ]                   [ 0  3  8  0  0  0  1  0  0 ]
      [ 7  7  7 ]                   [ 7  7  7  0  0  0  0  1  0 ]
1:    [ 2  5  9 ] ]          1:     [ 2  5  9  0  0  0  0  0  1 ]]
```

MINIMAL POLYNOMIAL OF A SQUARE MATRIX

'MPOL' calculates the minimal polynomial P of a square matrix A, i.e. the lowest-degree unit polynomial such that P(A)=0 (the polynomial P divides the characteristic polynomial of A. See program 'CARP' in the 'MATR' directory for further details).

The functional diagram is as follows:

```
      ┌─────────────┐      'MPOL'         ┌─────────────┐
  1:  │      A      │  ─────────────>  1: │ Expression  │
      └─────────────┘                     └─────────────┘
```

where "Expression" is an algebraic expression of the equation `P(A)=0`. If, for example, the minimal polynomial of A is `P = X^3 - 3*X + 2`, then the expression obtained is:

`'A^3-3*A+2*I=0'`

where the letter I denotes the identity matrix with the same format as A.

N.B: 'MPOL' calls 'ADDID', 'RANK' and 'GETR' in the 'MATR' directory, and 'ELML' in the 'POLY' directory.

'MPOL': (Checksum: # 24296d, Size: 341.5 bytes)

```
«  DUP  SIZE  1  GET  DUP  IDN
   →  a  n  b
   «  n  1  +  DUP  2  →LIST  0  CON
      n  1  +  DUP  SQ  n  -  FOR  i
         i  1  PUT
      n  STEP
      0  n  START
         b  OBJ→  DROP  'b'  a  STO*
      NEXT
      n  1  +  n  SQ  2  →LIST  →ARRY  *
      ADDID  RANK  DROP  n  1  +  GETR
      POLY  ELML  MATR  DUP  1  GET  /  OBJ→
      1  GET  1  +  0
      SWAP  3  FOR  i
         i  ROLL  'A'  i  2  -  ^  *  +
      -1  STEP
      SWAP  'I'  *  +  0  =
   »
»
```

Note: the coefficients obtained are real and likely to include round-off errors. If the coefficients of A are rational, then so are those of the minimal polynomial P (better still: if A has integer coefficients, then so does P).

It seems wise in such cases to force the approximation to produce rational values of the coefficients obtained using the instruction **→Q**, first switching to **n FIX** mode (with the integer simply serving to compensate for round-off errors).

PROGRAM 'MPOL': PRACTICAL EXAMPLES

Note:

The run time and space required in memory for 'MPOL' increase considerably as the order of the square matrix A for which we want to find the minimal polynomial increases.
Computation time is therefore approximately:
12 seconds for a 3x3 square matrix.
25 seconds for a 4x4 square matrix.

Example 1:

(Result obtained in **STD** mode)

```
    [[  3  -1   1  ]
     [ -1   3   1  ]      'MPOL'
1:   [  1   1   3  ]]   -------->   1:  'A^2-5*A+4*I=0'
```

Example 2:

(Result obtained with →**Q** in **8 FIX** mode)

```
    [[  1   1   1  ]
     [  1   1   1  ]      'MPOL'
1:   [  1   1   1  ]]   -------->   1:  'A^2-3*A=0'
```

Example 3:

(Result obtained in **STD** mode)

```
    [[  5   1   3  -2 ]
     [  4  -1   6  -4 ]
     [  3   1   1  -1 ]      'MPOL'        'A^4-8*A^3-14*A^2
1:   [ -2  -1  -1   3 ]]   -------->   1:  +50*A+3*I=0'
```

Example 4:

(Result obtained in **STD** mode)

```
    [[-142   21 -217  213 ]
     [ 300  -47  463 -451 ]
     [ 190  -30  293 -284 ]      'MPOL'
1:   [  68  -12  107 -102 ]]   -------->   1:  'A^4-2*A^3+A^2=0'
```

GAUSSIAN PIVOT METHOD

'PIVOT' allows you to apply the Gaussian elimination or "pivot" method to a matrix A. Two options are open to the user, depending on how far you wish to pursue the method:

First option:

Each pivot found eliminates all the terms below it.
Matrix A will therefore be gradually transformed into a matrix whose coefficients below the leading diagonal are zero (or more precisely, all the coefficients whose row number is higher than their column number).
This method is useful for determining the rank of a matrix, or for transforming a system of linear equations into a "cascading" system.

Second option:

Each pivot found eliminates all the terms above and below it.
Wherever a pivot has been used in a column it therefore remains the only non-zero element.
This method is used to invert a square matrix A (A will need to be bounded first to the right by the identity matrix, using program 'ADDID').
We can therefore use the method to solve a system of equations by making the system matrix a diagonal matrix.

Let us suppose that A is a matrix in the format {n,p} (n rows, p columns). At the kth step in the method, the pivot used is the term at the intersection of the kth row and the kth column.
The user will have to decide whether to swap rows (program 'SWPR') if ever the coefficient to be used as the pivot is zero (in which case you will be prompted by a message).

There are two functional diagrams for the two options described above:

First option:

```
                                  (program halted)
2:                     ---->   2:        B
1:      Matrix A               1:       List
```

(Here we simply eliminate the coefficients below the diagonal).

Second option:

```
                                  (program halted)
2:      Matrix A       ---->   2:        B
1:         1                   1:       List
```

(Here all coefficients on either side of the pivot in the same column will be eliminated).

'PIVOT' halts at each step in the calculation, i.e. once a pivot has been used to its full extent.

The various operations done using the pivot are then listed in the form of algebraic expressions.

To take an example, the expression:

```
'R4=2*R4-3*R2'
```

means that the 4th row (denoted R4) has been replaced by another row found using the following operation:
"twice row R4 - three times row R2".

This means that the pivot is in the 2nd row and that, for example, rows 2 and 4 of the matrix were in the following form:

```
R1 -> [[ *  *  *  *  *   ..... ]
R2 ->  [ 0  2  *  *  *   ..... ]
R3 ->  [ 0  *  *  *  *   ..... ]
R4 ->  [ 0  3  *  *  *   ..... ] .
        ........ etc
```

The operation described above therefore eliminates the coefficient equal to 3 in row R4.

The program halts after each step in the method, so we therefore get a list of all operations done at level 1 and the modified matrix at level 2.
You simply have to copy the list and the modified matrix onto a piece of paper before pressing DROP (to delete the list) then CONT to continue the program and move on to the next pivot. The program terminates once all the steps in the method have been completed.

Note:
'PIVOT' constantly tests for integer coefficients in the matrix in question.
If required, it will perform simplifications while eliminating by calling program 'SIMP' in the 'ARIT' directory (the aim being to simplify all operations as far as possible).

Example:
In the case below, the operation used to eliminate the coefficient 6 in R4 is:

```
'L4 = 4*L4 - 3*L2'
```

```
R1 -> [[ *  *  *  *  *   ..... ]
R2 ->  [ 0  8  *  *  *   ..... ]
R3 ->  [ 0  *  *  *  *   ..... ]
R4 ->  [ 0  6  *  *  *   ..... ]
          ........ etc
```

(which is simpler than `'R4 = 8*R4 - 6*R2'`).

Note:

Program 'PIVOT' calls program 'SIMP' in the 'ARIT' directory.

TEXT OF PROGRAM 'PIVOT'

'PIVOT': (Checksum: # 30359d, Size: 516.5 bytes)

```
«  DUP   TYPE   NOT   DUP   DROPN   OVER   SIZE   EVAL
   «   RCLF   STD   " 'R"   ROT   →STR   +   OBJ→   SWAP   STOF   »

   →   f   r   c   R

   «   1   r   f   NOT   -   c   MIN   FOR   j
         WHILE   DUP   j   j   2   →LIST   GET   NOT
         REPEAT   "Zero pivot"   HALT   END
         DUP   j   GETR   DUP   j   GET

         →   Rj   p

         «   { }
             1   f   NOT   j   *   +   r   FOR   i
                 IF   i   j   ≠   THEN
                     OVER   i   GETR   DUP   j   GET
                     IF   DUP   THEN
                         p
                         IF   DUP   0   <   THEN
                             R→C   NEG   C→R
                         END
                         IF   OVER   FP   NOT   OVER   FP   NOT   AND
                             THEN   ARIT   SIMP   MATR
                         END
                         DUP2   SWAP   i   R   EVAL   ROT   OVER   *
                         ROT   j   R   EVAL   *   -   =
                         5   ROLL   SWAP   +   5   ROLLD
                         ROT   *   Rj   ROT   *   -   i   PUTR   SWAP
                     ELSE
                         DROP2
                     END
                 END
             NEXT
         »

         HALT

      NEXT
   »

»
```

PROGRAM PIVOT: PRACTICAL EXAMPLE

Example 1:

We want to find the inverse of the matrix:

```
    [[ 4 3 8 ]
A =  [ 2 5 1 ]
     [ 6 3 2 ]]
```

We put matrix A at level 1. We then call program 'ADDID' to bound matrix A to the right with the identity matrix of order 3.

We enter a value of 1 at level 1, the matrix is shifted up to level 2, and we call program 'PIVOT'. The following results are obtained on the stack (to move on to the next step, delete the list at level 1 before pressing CONT):

```
    [[ 4  3   8  1  0  0 ]
     [  0  7  -6 -1  2  0 ]
2:   [  0 -3 -20 -3  0  2 ]]

       { 'R2=2*R2-R1'
1:       'R3=2*R3-3*R1' }
```

```
    [[28 0  74    10 -6  0 ]
     [ 0 7  -6    -1  2  0 ]
2:   [ 0 0 -158  -24  6 14 ]]

     { 'R1=7*R1-3*R2'
1:       'R3=7*R3+3*R2' }
```

```
    [[   2212  0     0 -98 -252  518 ]
     [    0  553     0  -7  140  -42 ]
2:   [    0     0 -158 -24    6   14 ]]

          { 'R1=79*R1+37*R3'
1:          'R2=79*R2-3*R3' }
```

All we then need to do is divide the first row by 2212, the second by 553 and the third by -158 to obtain the expression of the inverse of the initial matrix A in the right-hand part of the array.

For line 1, for example, do (having first deleted the list):

DUP 1 GETR 2212 / 1 PUTR

More precisely, using program 'A→Q' in the 'R.C' directory on each row, we find:

```
         1   [[-7 -18  37]
A^(-1) = ——   [-2  40 -12]
        158   [24  -6 -14]]
```

PROGRAM PIVOT: PRACTICAL EXAMPLE

Example 2:

We want to calculate the rank of the following family of vectors:

```
U1 = [ 1  5  9 13 17 ],
U2 = [ 3  7 11 15 19 ],
U3 = [ 2  6  1  0 11 ],
U4 = [ 1  3 14 21 16 ].
```

We put the matrix:

```
        [[ 1  5  9 13 17 ]
         [ 3  7 11 15 19 ]
  A =    [ 2  6  1  0 11 ]
         [ 1  3 14 21 16 ]]
```

at level 1 and call program 'PIVOT'

We obtain the following steps:

```
   [[ 1   5   9    13   17 ]
    [ 0  -8 -16   -24  -32 ]
    [ 0  -4 -17   -26  -23 ]
2:  [ 0  -2   5     8   -1 ]]

     {  'R2=R2-3*R1'
        'R3=R3-2*R1'
1:      'R4=R4-R1'
```

```
   [[ 1   5   9    13   17 ]
    [ 0  -8 -16   -24  -32 ]
    [ 0   0  18   28   14 ]
2:  [ 0   0 -36   -56  -28 ]]

      { 'R3=2*R3-R2'
        'R4=4*R4-R2' }
1:
```

```
   [[ 1   5   9    13   17 ]
    [ 0  -8 -16   -24  -32 ]
    [ 0   0 -18  -28  -14 ]
2:  [ 0   0   0     0    0 ]]
1:       { 'R4=R4+2*R3' }
```

The last step shows that the family of vectors U1, U2, U3 and U4 has a rank of 3 (triangulation finishes with precisely three non-zero pivots).

FINDING LINEAR RELATIONS OR EQUATIONS

'EQLR' is a program designed to find the linear equations of a vector space, or the linear relations that may exist between two given vectors.

There are two functional diagrams for this program, depending on what you want to do with it:

```
     ┌──────────────┐                      ┌─────────────────┐
  2: │    Matrix    │       'EQLR'         │Equations (if 1) │
     ├──────────────┤  ───────────────>    │       or        │
  1: │    1 or 2    │                   1: │Relations (if 2) │
     └──────────────┘                      └─────────────────┘
```

'EQLR' calls programs 'ADDID' and 'RANK'.

See the practical examples on the following pages for further details.

'EQLR' : (Checksum: # 32602d, Size: 307 bytes)

```
«  {  « TRN   "'X"    »    "'U" }    SWAP    GET    EVAL
   OVER   SIZE   EVAL    →   s    n   p
   «   DUP   ADDID   1   RANK   ROT   RANK   SWAP   DROP   -
       «   RCLF   STD   s   ROT   →STR   +   OBJ→   SWAP   STOF   »
       →   m   u
       «   { }   IF   m   THEN
                 n   m   -   1   +   n   FOR   i
                     0   1   n   FOR   j
                         3   PICK   i   p   j   +   2   →LIST
                         GET   j   u   EVAL   *   +
                     NEXT   0   =   +
                 NEXT
           END   SWAP   DROP
       »
   »
»
```

Note:

The approximation of the coefficients in the relations found can be forced to give rational numbers using the instruction →**Q** (switching first to **n FIX** mode in order to prevent round-off errors).

PROGRAM 'EQLR': FINDING LINEAR EQUATIONS

Let us take the following problem: given n vectors U_1, U_2, ..., U_n all having p coefficients (therefore vectors of $\mathbb{R}^p$, do the vectors form a family that generates $\mathbb{R}^p$? If not, what are the linear equations of the vector subspace of $\mathbb{R}^p$ that they generate?
The dimension r of this vector subspace is called the family's rank:

If r=p, the vectors U_k are generators in $\mathbb{R}^p$.

If r<p, the vector subspace they generate is defined by a system of p-r linearly independent equations.

Program 'EQLR' deals with this problem as shown in the functional diagram below:

```
2: | Matrix |
                 'EQLR'
1: |   1    |  ---------->   1: | List |
```

'Matrix' is a matrix whose n rows are the n vectors U_k (each with p coefficients).
'List' is a list of equations found:
This list is empty if the vectors U_k generate all of $\mathbb{R}^p$.
If the system of n vectors U_k has a rank r<p, the list includes the p-r linear independent equations defining the subspace generated by the vectors U_k.
Each equation is written in the form of an algebraic expression linking the coordinates denoted by X1, X2, etc.

<u>Example</u>:

We want to determine the subspace of $\mathbb{R}^5$ generated by the vectors:

```
U1 = [ 1 5  9 13 17 ]   U2 = [ 3 7 11 15 19 ]
U3 = [ 2 3  4  5  6 ]   U4 = [ 3 9 15 21 27 ]
```

We therefore put the following matrix at level 2:

```
      [ [ 1 5  9 13 17 ]
        [ 3 7 11 15 19 ]
A =     [ 2 3  4  5  6 ]
        [ 3 9 15 21 27 ] ]
```

And put the integer 1 at level 1 then call 'EQLR'.

After 32 seconds, we find the following at level 1 of the stack:

```
{    '-(.75*X1)+3*X4-2.25*X5=0'
     '-(.66666667*X2)+2*X4-1.33333333*X5=0'
     '-(.5*X3)+X4-.5*X5=0'    }
```

These three relations can be written in simpler form:

```
X1 - 4*X4 + 3*X5 = 0,  X2 - 3*X4 + 2*X5 = 0,  X3 - 2*X4 + X5 = 0.
```

Which proves that the system of vectors U_k has a rank of 2, since 3 linearly independent equations are needed to characterize the vector subspace of $\mathbb{R}^5$ that they generate.

PROGRAM 'EQLR': FINDING LINEAR RELATIONS

Let us take the following problem: given n vectors U_1, U_2, ..., U_n, are there any non-trivial linear relations between the vectors, i.e. relations of the type: $a_1U_1 + a_2U_2 + ... + a_nU_n = 0$ where the scalars a_k are not all equal to zero?

Program 'EQLR' deals with this question by giving (if the vectors are related) the largest possible number of independent relations between the vectors (if the n vectors satisfy k linearly independent relations, they form a system with a rank of n-k).

The functional diagram is as follows:

```
2: | Matrix |         'EQLR'
1: |   2    |   ---------------->   1: | List |
```

'Matrix' is a matrix whose rows are the vectors U_k.
'List' is the list of relations found:
The list is empty if the vectors U_k are free.
If the system of n vectors U_k has a rank of n-k, the list includes k linearly independent relations found between the vectors U_k.
Each of these relations is an algebraic expression in which the vectors are denoted by U1, U2, etc. according to the row in which they appear in the initial matrix.

Example:

We want to know if there are any linear relations between the vectors:

```
U1 = [ 1 5  9 13 17 ]  U2 = [ 3 7 11 15 19 ]
U3 = [ 2 3  4  5  6 ]  U4 = [ 3 9 15 21 27 ]
```

We therefore put the following matrix at level 2:

```
      [ [ 1 5  9 13 17 ]
        [ 3 7 11 15 19 ]
  A =   [ 2 3  4  5  6 ]
        [ 3 9 15 21 27 ] ]
```

And put the integer 2 at level 1 then call 'EQLR'.

After 20 seconds, we find the following list at level 1 of the stack:

```
{ '-(.75*U1)-.5*U3+.58333333*U4=0'
  '-(1.5*U2)+U3+.83333333*U4=0' }
```

The two relations found can be written more simply:

```
-9*U1 - 6*U3 + 7*U4 = 0,  et -9*U2 + 6*U3 + 5*U4 = 0.
```

We can therefore deduce that U1, U2, U3 and U4 are related. More precisely, they form a system with a rank of 2.

ANALYSIS

This somewhat vague heading covers the programs in the 'ANLY' directory, which can be used to solve standard types of problems such as:

* the equation of the tangent to a given point on a curve Y=F(X).
* finding points on a curve Y=F(X) where the tangent is horizontal.
* finding the points of inflection of a curve Y=F(X).
* approximation of an application using the least squares method.
* interpolation by Lagrange polynomial.
* approximate numerical resolution of non-linear systems.
* partial sums of Fourier series.

A lot of other programs in other directories also use analytical techniques. The programs here are those which do not clearly belong in a more specialized directory.

Here is the list of programs in the 'ANLY' directory:

'TNGT'	:	Equation of the tangent to a curve Y=F(X) at a given point.
'EXTRE'	:	Finding the local extreme points of the function Y=F(X).
'INFL'	:	Finding the points of inflection on the curve Y=F(X).
'LSAP'	:	Least squares approximation, using linear combinations of free functions, of a function f whose values are known at a certain number of points.
'LAGR'	:	Calculating the Lagrange interpolation polynomial through a family of given points.
'PLOT'	:	Plotting of the family of points used in programs 'LSAP' and 'LAGR', and of the curve obtained.
'SXY'	:	Approximate numerical resolution, using Newton's method, of a system of 2 equations in 2 unknowns: `F(X,Y)=0, G(X,Y)=0`
'SXYZ'	:	Approximate numerical resolution, using Newton's method, of a system of 3 equations in 3 unknowns: `F(X,Y,Z)=0, G(X,Y,Z)=0, H(X,Y,Z)=0`
'RK4'	:	Approximate resolution, using the 4th-order Runge-Kutta method, of the differential equation Y'=F(X,Y).
'FOUR'	:	Calculating the partial sums of the Fourier series of a given periodic function.

TANGENT TO A CURVE Y=F(X)

'TNGT' gives the equation of the tangent at the point x = α to the curve `Y=F(X)`, i.e. the straight line whose equation is:

$$Y = (X-\alpha)f'(\alpha) + f(\alpha)$$

This equation is obtained in the form `'Y=a*X+b'`.

The functional diagram is as follows:

2:	F(X)	'TNGT'	2:	
1:	α	——————>	1:	equation

where F(X) is the expression that characterizes the function F (a capital X must be used) and α is a real number (the x-coordinate of the point of tangency).

'TNGT': (Checksum: # 8951d, Size: 131 bytes)

```
«   'X'   STO   DUP   'X'   ∂   SWAP   EVAL
    →   df   f
    «   'Y'   df   'X'   *   f   X   df   *   -   +   =
        'X'   PURGE
    »
»
```

Example:

Tangent to the curve y=LN(1+X²) at the point x=3.
In 3 FIX mode, we obtain:

2:	'LN(1+X*X)'	'TNGT'	2:	
1:	3	——————>	1:	'Y=0.600*X+0.503'

POINTS ON THE CURVE Y=F(X) WHERE THE TANGENT IS HORIZONTAL

==================================

'EXTRE' lets you find the characteristics of a point on the curve `Y=F(X)` where the tangent is horizontal.

The functional diagram is as follows:

```
    ┌──────────────┐                    ┌───────────────┐
2:  │     F(X)     │      'EXTRE'   3:  │      X: ..... │
    ├──────────────┤   ───────────> 2:  │      F: ..... │
1:  │      a       │                1:  │    F'': ..... │
    └──────────────┘                    └───────────────┘
```

At level 2, F(X) is the expression of the function F (in terms of the variable X) and a is a numerical value with which to start the program. The value of a must not be too far away from the solution we want.

'EXTRE' uses the **ROOT** instruction to look for a value X to cancel out the derivative F'.

If it finds one, the program gives:

* the value of X (real number denoted by "X" at level 3).
* the value of F(X) (real number denoted by "F" at level 2).
* the value of the second derivative F'' at the point X (in the form of a real number denoted by "F''").

If `F''(X) < 0`, we find a local maximum.
If `F''(X) > 0`, we find a local minimum.
If `F''(X) = 0`, (neglecting round-off errors), we usually find a point of inflection where the tangent is horizontal.

'EXTRE': (Checksum: # 51935d, Size: 166.5 bytes)

```
«  'X'   PURGE   OVER   'X'   ∂   DUP   'X'   ∂
   →   f   a   df   ddf
   «   df   'X'   a   ROOT   "X"   →TAG
       f   EVAL   "F"   →TAG
       ddf   EVAL   "F''"   →TAG   'X'   PURGE
   »
»
```

Example: in 12 seconds in "5 FIX" mode:

```
    ┌──────────────┐                 ┌──────────────────┐
2:  │ 'X*EXP(-√X)' │   'EXTRE'   3:  │      X:  4.00000 │
    ├──────────────┤   ───────>  2:  │      F:  0.54134 │
1:  │      1       │             1:  │    F'': -0.01692 │
    └──────────────┘                 └──────────────────┘
```

The point `(4,≈0.54134)` therefore represents a maximum value.

POINTS OF INFLECTION ON THE CURVE Y=F(X)

'INFL' lets you find the characteristics of a point with a zero second derivative (usually a point of inflection) on the curve Y=F(X).

The functional diagram is as follows:

```
         +-----------+                    +----------------+
    2:   |   F(X)    |      'INFL'    3:  |      X:  ..... |
         +-----------+    -------->   2:  |      F:  ..... |
    1:   |     a     |                1:  |     F':  ..... |
         +-----------+                    +----------------+
```

At level 2, F(X) is the expression of the function F (in terms of the variable X) and a is a numerical value with which to start the program. The value of a must not be too far away from the solution we want.

'INFL' uses the **ROOT** instruction to look for a value X to cancel out the second derivative F".

If it finds one, the program gives:
* the value of X (real number denoted by "X" at level 3).
* the value of F(X) (real number denoted by "F" at level 2).
* the value of the derivative F' at the point X (in the form of a real number denoted by "F'".

If `F'(X)` ≠ `0`, we find a point of inflection.
If `F'(X)` = `0`, (neglecting round-off errors), we cannot be sure that the point is a point of inflection.

'INFL': (Checksum: # 16137d, Size: 164.5 bytes)

```
«  'X'   PURGE   OVER   'X'   ∂   DUP   'X'   ∂
   →   f   a   df   ddf
   «   ddf   'X'   a   ROOT   "X"   →TAG
       f   EVAL   "F"   →TAG
       df   EVAL   "F'"   →TAG   'X'   PURGE
   »
»
```

<u>Example</u>: in 18 seconds in "5 FIX" mode

```
         +---------------+                    +--------------------+
    2:   | 'X*EXP(-√X)'  |     'INFL'     3:  |       X:  9.00000  |
         +---------------+   -------->    2:  |       F:  0.44808  |
    1:   |       6       |                1:  |      F': -0.02489  |
         +---------------+                    +--------------------+
```

The point `(9,≈0.44808)` is therefore a point of inflection.

LEAST SQUARES APPROXIMATION

Let (X1,Y1), (X2,Y2),...., (Xi,Yi) be n points in the plane (of distinct pairs of x-coordinates).
Let p be a family of linearly independent functions F1, F2,..., Fi,..., Fp, where $p < n$.

This gives us a single application F, which is a linear combination of Fk, to find the best approximation of the points (Xi,Yi) using the least squares method, i.e. to minimize the sum:

$$\sum_{i=1}^{i=n} (Yi - F(Xi))^2 .$$

For program 'LSAP' to run correctly (and thus to determine the solution of F), we must first:

* enter for a variable called 'DATA' a **vector** formed by the points (Xi,Yi) and written as complex numbers.

example: `'DATA' <--  [ (-2,3) (-1,0) (0,1) (1,5) (2,3) ]`

* enter for a variable called 'FUNC' the list of parameters characterizing the functions Fk.

  The format of 'FUNC' must be as follows:
  { Fk start step number } where:
  "Fk" is the algebraic expression of the function Fk.
  "start" is the minimum value of the integer variable k.
  "step" is the increment of the integer variable k.
  "number" is the number of different values of k.

Example: `{'X^K' 0 1 5}` if we want to find the solution of F as a linear combination of the functions `1, X, X^2, X^3` and `X^4`.

Note:Capital 'K' and 'X' must be used in the expression.

The stack is not affected by program 'LSAP', which places the solution F as an algebraic expression in the variable 'EQ'. The result can thus be displayed using the DRAW instruction or evaluated by SOLVR.

TEXT AND PRACTICAL EXAMPLE OF PROGRAM 'LSAP'

'LSAP': (Checksum: # 16447d, Size: 348.5 bytes)

```
«  DATA   C→R    DUP    SIZE   1    GET    FUNC    EVAL
   →   a    b    c    f    m    p    L
   «   1    L    FOR   i
            m    i    1    -    p    *    +    'K'    STO
            1    c    FOR   j
                 a    j    GET    'X'    STO    f    EVAL
            NEXT
       NEXT
       L   c   2   →LIST   →ARRY   DUP   DUP   TRN   *
       MATR   DIVAC   ANLY   b   *   'X'   PURGE
       0
       1    L    FOR   i
            OVER   i   GET   m   i   1   -   p   *   +
            'K'   STO   f   EVAL   *   +
       NEXT
       STEQ   DROP   'K'   PURGE
   »
»
```

N.B: Program 'LSAP' calls program 'DIVAC' in directory 'MATR'.

Example:

We enter the vector [(-3,-1) (-2,1) (-1,2) (1,2) (2,1) (3,0)]
for the variable 'DATA'.

We then look for the application capable of finding the best approximation (using the least squares method) of the cluster of points (-3,-1), ..., (3,0), written as follows:

F(X)= a + b*cos(X) + c*cos(2*X) + d*cos(3*X) + e*cos(4*X).

We therefore enter the list { 'COS(K*X)' 0 1 5) for the variable 'FUNC'.

We then call 'LSAP', which runs for 12 seconds.
'LSAP' puts the following algebraic expression into the variable 'EQ' (in 3 FIX mode):

'1.030+0.401*COS(X)+0.349*COS(2*X)+0.140*COS(3*X)-1.588*COS(4*X)'

Evaluating 'EQ' gives the following results:

X	-3	-2	-1	1	2	3
F(X)	-0.500	1.000	2.000	2.000	1.000	-0.500

(at -2, -1, 1, 2, the values obtained are correct to 1E-11).

Note:the families of functions most often used are:

Fk(X)=cos(kX), Fk(X)=sin(kX), Fk(X)=X^K, Fk(X)=exp(kX).

CALCULATING THE LAGRANGE INTERPOLATION POLYNOMIAL

Let (X1,Y1), (X2,Y2),...., (Xi,Yi), ..., (Xn,Yn) be n points in the plane (of distinct pairs of x-coordinates).
This gives us a single polynomial P of degree < n such that for any index i, P(Xi)=Yi. We say that P is the Lagrange interpolation polynomial corresponding to the cluster of points (Xi,Yi).

'LAGR' computes this polynomial and enters it for the variable 'EQ' as an algebraic expression. We can then plot the polynomial using DRAW or evaluate it using SOLVR.
To do this, we first have to enter the vector [(X1,Y1) (X2,Y2), ..., (Xn,Yn)] whose elements are complex numbers representing the points (Xi,Yi) for the variable 'DATA'.

'LAGR': (Checksum: # 17449d, Size: 221.5 bytes)

```
«  DATA   C→R   SWAP   DUP   SIZE   1   GET
   →  a  n
   «  1  n  FOR  i
            a  i  GET
            0  n  1  -  FOR  j
                 DUP  j  ^  SWAP
            NEXT
            DROP
         NEXT
         n  n  2  →LIST  →ARRY  MATR  DIVAC  ANLY
         0
         1  n  FOR  i
            OVER  i  GET  'X'  i  1  -  ^  *  +
         NEXT
         STEQ  DROP
   »
»
```

N.B: Program 'LAGR' calls program 'DIVAC' in the 'MATR' directory.

Example:

Having first entered [(-3,3) (-2,1) (-1,2) (1,3) (2,-1) (3,2)]

for 'DATA' and run 'LAGR', we find in 'EQ' (in 5s):

```
'4 + 1.03333333333*X - 1.66666666667*X^2 - .583333333333*X^3
   +.166666666667*X^4+.05*X^5'.
```

This polynomial confirms:

```
P(-3)=3,  P(-2)=.99999999999, P(-1)=2, P(1)=3,
P(2)=-1.00000000001, P(3)=2.
```

PLOTTING POINTS AND THE CURVE APPROXIMATING TO THEM

The variable 'DATA' is assumed to be a vector of complex numbers `(Xi,Yi)`. These complex numbers represent a cluster of points in the plane.

The variable 'EQ' is assumed to be an algebraic expression (or program) that can be plotted on the display by DRAW.

'PLOT' then plots the following:

* the points of the variable 'DATA'.
* the expression (or program) 'EQ'.

N.B: The points of the variable 'DATA' in fact appear on the display as small squares to make them easier to see.
The dimensions of PICT and the plotting intervals for the X and Y-axes are not changed.

Program 'PLOT' is specially designed to be called after programs 'LAGR' (finding the Lagrange interpolation polynomial) and 'LSAP' (least squares approximation).

We can thus display the cluster of points `(Xi,Yi)` and the curve approximating to it.

Once all points have been plotted, we return to the GRAPH environment.

We then press 'ON' to quit the program.

'PLOT': (Checksum: # 60176d, Size: 202 bytes)

```
«   { # 1d  # 1d }    PX→C    {# 0d # 0d }    PX→C    -
    →   d
    «   ERASE    { # 0d  # 0d }    PVIEW
        DATA    OBJ→    1    GET
        1    SWAP    START
             DUP    d    -    SWAP    d    +    BOX
        NEXT
        FUNCTION    'X'    INDEP    DRAW    GRAPH
    »
»
```

SOLVING A SYSTEM OF EQUATIONS IN TWO UNKNOWNS BY ITERATION

'SXY' lets you find an approximate solution, using Newton's method, of a system of two equations in two unknowns X and Y:

$$F(X,Y)=0, \quad G(X,Y)=0$$

The principle is as follows:
We define a sequence (Xn,Yn) starting from an initial point (X0,Y0) and the relation:

$$\begin{bmatrix} X(n+1) \\ Y(n+1) \end{bmatrix} = \begin{bmatrix} X(n) \\ Y(n) \end{bmatrix} - \begin{bmatrix} J(\ X(n),Y(n)\ ) \end{bmatrix}^{-1} \begin{bmatrix} F(\ X(n),Y(n)\ ) \\ G(\ X(n),Y(n)\ ) \end{bmatrix}$$

where J is the Jacobian matrix:

$$J(X,Y) = \begin{bmatrix} F'_x\ (X,Y) & F'_y\ (X,Y) \\ G'_x\ (X,Y) & G'_y\ (X,Y) \end{bmatrix}$$

In certain conditions, the sequence (Xn,Yn) converges to a solution to the system.

The stack should look like this at the start:

2:	F(X,Y)
1:	G(X,Y)

F(X,Y) and G(X,Y) are the algebraic expressions of the applications F and G (in which a capital X and Y must be used).

We then call program 'SXY'. The program halts after a short time (as the partial derivatives are computed) and a menu is displayed with the entries **NEW** (on the left) and **EXIT** (on the right).

We then enter the starting point [X0, Y0] for iteration, in vector form, at level 1 of the stack.

By pressing the **NEW** key we obtain [X1, Y1]., then [X2, Y2], etc.

Press the **EXIT** key to quit the program.

Whenever the program halts, you can enter a new starting point [X0, Y0] at level 1 of the stack (in place of the point [Xn, Yn] just obtained) if you want to do another search before pressing **NEW**.

TEXT OF PROGRAM 'SXY' AND PRACTICAL EXAMPLE

'SXY': (Checksum: # 58688d, Size: 347.5 bytes)

```
«   →   f  g
    «   { X Y }    PURGE
        f    'X'   ∂   f   'Y'   ∂   g   'X'   ∂   g   'Y'   ∂
        →    dfx   dfy   dgx   dgy
        «    {
                 {     "NEW"
                       «   DUP   DUP   V→   'Y'   STO   'X'   STO
                           dfx  EVAL  dfy  EVAL  dgx  EVAL  dgy  EVAL
                           { 2 2 }   →ARRY   INV   f   EVAL   g   EVAL
                           2   →ARRY   *   -
                       »
                 }
                 {}   {}   {}   {}
             {   "EXIT"        CONT        }
             }
             TMENU   HALT   { X Y }   PURGE   2   MENU
        »
    »
»
```

Example:

We want to solve the system $X^2+Y^2=1$, $2X+Y=1$.
We therefore create the stack shown below and call 'SXY'.

2:	'X*X+Y*Y-1'
1:	'2*X+Y-1'

The menu displaying the "NEW" and "EXIT" entries appears after three seconds. We then enter the starting point, [1,0] for example, at level 1 and press "NEW".
The point [1,-1] is then entered into the stack after a few seconds. A new point is displayed (computed within one second) each time you press NEW. In 6 FIX mode we therefore find:

```
      [ 0.833333  -0.666667 ],  [ 0.801282  -0.602564 ],
  and [ 0.800002  -0.600004 ], etc...
```

The sequence (Xn,Yn) converges to [0.8, -0.6], which gives us a solution to the problem.

We then press "EXIT" to quit the program.

SOLVING A SYSTEM OF EQUATIONS IN THREE UNKNOWNS BY ITERATION

'SXYZ' lets you find an approximate solution, using Newton's method, of a system of three equations in three unknowns X, Y and Z:

$$\begin{cases} F(X,Y,Z)=0 \\ G(X,Y,Z)=0 \\ H(X,Y,Z)=0 \end{cases}$$

The principle is as follows:

We define a sequence `(Xn,Yn,Zn)` starting from an initial point `(X0,Y0,Z0)` and the relation:

$$\begin{bmatrix} X(n+1) \\ Y(n+1) \\ Z(n+1) \end{bmatrix} = \begin{bmatrix} X(n) \\ Y(n) \\ Z(n) \end{bmatrix} - \Big[\, J(\ X(n),Y(n),Z(n)\ )\,\Big]^{-1} \begin{bmatrix} F(X(n),Y(n),Z(n)) \\ G(X(n),Y(n),Z(n)) \\ H(X(n),Y(n),Z(n)) \end{bmatrix}$$

where J is the matrix:

$$J(X,Y,Z)) = \begin{bmatrix} F_x'\ (X,Y,Z) & F_y'\ (X,Y,Z) & F_z'\ (X,Y,Z) \\ G_x'\ (X,Y,Z) & G_y'\ (X,Y,Z) & G_z'\ (X,Y,Z) \\ H_x'\ (X,Y,Z) & H_y'\ (X,Y,Z) & H_z'\ (X,Y,Z) \end{bmatrix}$$

In certain conditions, the sequence (Xn,Yn,Zn) converges to a solution to the system.

The stack should look like this at the start:

3:	F(X,Y,Z)
2:	G(X,Y,Z)
1:	H(X,Y,Z)

`"F(X,Y,Z)"`, `"G(X,Y,Z)"` and `"H(X,Y,Z)"` are the expressions of the applications F, G and H (in which a capital X, Y and Z must be used).

We then call program 'SXYZ'. The program halts after a short time (as the partial derivatives are computed) and a menu is displayed with the entries **NEW** (on the left) and **EXIT** (on the right).

We then enter the starting point [X0, Y0, Z0] for iteration, <u>in vector form</u>, at level 1 of the stack.

By pressing the **NEW** key we obtain `[ X1, Y1, Z1 ].`, then `[ X2, Y2, Z2 ]`, etc.

Press the **EXIT** key to quit the program.

Whenever the program halts, you can enter a new starting point `[ X0, Y0, Z0 ]` at level 1 of the stack (in place of the point `[ Xn, Yn, Zn ]` just obtained) if you want to do another search before pressing **NEW**.

TEXT OF PROGRAM 'SXYZ' AND PRACTICAL EXAMPLE

'SXYZ': (Checksum: # 32791d, Size: 541 bytes)

```
«   →   f   g   h
    «   { X Y Z }   PURGE   f   'X'   ∂   f   'Y'   ∂   f   'Z'   ∂
        g   'X'   ∂   g   'Y'   ∂   g   'Z'   ∂
        h   'X'   ∂   h   'Y'   ∂   h   'Z'   ∂
        →   dfx   dfy   dfz   dgx   dgy   dgz   dhx   dhy   dhz
        «   {
                {   "NEW"
                    «   DUP   DUP   V→
                        'Z'   STO   'Y'   STO   'X'   STO
                        dfx   EVAL   dfy   EVAL   dfz   EVAL
                        dgx   EVAL   dgy   EVAL   dgz   EVAL
                        dhx   EVAL   dhy   EVAL   dhz   EVAL
                        {3 3}   →ARRY   INV   f   EVAL   g   EVAL
                        h   EVAL   3   →ARRY   *   -
                    »
                }
                { }   { }   { }   { }
                {   "EXIT"   CONT   }
            }
            TMENU   HALT   { X Y Z }   PURGE   2   MENU
        »
    »
»
```

Example: let us take the system $X^2+Y^2+Z^2=6$, $X+3Y+2Z=5$, $X^3+YZ=-1$.

We therefore create the stack shown below and call 'SXYZ':

3:	'X*X+Y*Y+Z*Z-6'
2:	'X+3*Y+2*Z-5'
1:	'X^3+Y*Z+1'

The menu displaying the "NEW" and "EXIT" appears. We then enter the starting point, `[3 2 0]` for example, at level 1.

A new point is displayed each time you press NEW. In 3 FIX mode we therefore find: `[ 2.041 1.689 -1.053 ]`, and `[1.427 1.846 -.0982]`, `[1.111 1.967 -1.006] etc.`
We soon arrive at `[ 1 2 -1 ]`.

The sequence (X_n, Y_n, Z_n) converges to $[1,2,-1]$, which gives us a solution to the problem.

We then press "EXIT" to quit the program.

SOLVING A DIFFERENTIAL EQUATION Y'=F(X,Y)

'RK4' allows you to find an approximate solution to a differential equation `Y'=F(X,Y)` (a first-order equation solved for Y').
The method used is the fourth-order Runge-Kutta method.

The principle is as follows:
We want to find the values of the solution f to Cauchy's problem:

```
f'(x) = F( x, f(x)), f(x0) = y0
```

where (x0,y0) is a given point (a unique solution can be found, if we admit certain assumptions about the regularity of F).

We define the sequence `Xn = X0 + n*h`, where h is the "step" of the method, and the sequence Yn defined by its initial term is Y0 and by the system:

```
a = h * F( X(n-1),Y(n-1) ).
b = h * F( X(n-1) + h/2 , Y(n-1) + a/2 ).
c = h * F( X(n-1) + h/2 , Y(n-1) + b/2 ).
d = h * F( X(n-1) + h , Y(n-1) + c ).
```

and `Y(n) = Y(n-1) + ( a + 2*b + 2*c + d ) / 6.`

Y(n) are therefore approximate values of `F(X(n))=F(X0+n*h)`.

Before calling 'RK4', we have to enter the expression `F(X,Y)` (capital X and Y must be used) at level 1.
We then start 'RK4'. The program halts after a short time (once the partial derivatives have been computed) and the following menu is displayed:

NEW		->N	->STEP		EXIT

At the same time the real numbers denoted by "STEP: 0.05" and "N: 1" are entered at level 2 and 1, thus specifying that the default value for the step of the method (the number h) is 0.05 and that one point will be obtained at a time.

The step can be changed by entering a new value at level 1 and pressing the "→STEP" key.
We can also ask for n points each time by entering an integer n at level 1 and pressing the "→N" key.

Each time you press the "NEW" key, n points (default value n=1) are computed and displayed.

All points are given in vector form i.e. `[ X, Y ]`.

Press the "EXIT" key to quit the program.

TEXT AND PRACTICAL EXAMPLE OF PROGRAM 'RK4'

'RK4': (Checksum: # 55009d, Size: 732 bytes)

```
« [ 0 0 ] .05  1  0  0  0  0
  →  f  v  h  n  k1  k2  k3  k4
  «  .05  "STEP"  →TAG  1  "N"  →TAG
     {
        {  "NEW"
           «  DUP  DUP  'v'  STO  V→  'Y'  STO  'X'  STO
              1  n  START
                 f  →NUM  h  *  'k1'  STO  'X'  h  2  /  STO+
                 'Y=v(2)+k1/
2'  DEFINE  f  →NUM  h  *  'k2'  STO
                 'Y=v(2)+k2/2'  DEFINE  f  →NUM  h  *  'k3'  STO
                 'X'  h  2  /  STO+
                 'Y=v(2)+k3'  DEFINE  f  →NUM  h  *  'k4'  STO
                 X  'v(2)+(k1+2*k2+2*k3+k4)/6'  →NUM
                 2  →ARRY  DUP  'v'  STO
              NEXT
           »
        }  { }  {  "→N"  «  'n'  STO  »  }
        {  "→STEP"  «  'h'  STO  »  }  { }  {  "EXIT"  CONT  }
     }
     TMENU  HALT  {X Y}  PURGE  2  MENU
  »
»
```

Example:

We want to find the solution f of the equation `Y'=2*X*(1+Y^2)` equal to zero at the origin.
This solution is given by `f(X)=TAN(X^2)`.
We enter `'2*X*(1+Y^2)` at level 1 and then call 'RK4'.
The menu is displayed with the entries "NEW", "→N", "→STEP" and "EXIT", along with the real numbers denoted by "STEP: 0.05" and "N: 1" at levels 2 and 1.

We enter the starting point [0, 0] at level 1 of the stack and press "NEW".

We then obtain the next point (in 3 FIX mode: `[0.050, 0.003]`).
To save time, we enter 10 at level 1 and press the "→N" key.
Pressing "NEW" again, we obtain the next 10 points:

For example, in STD mode:

`[.2  4.00235117567E-2]`, the exact value being:
`tan(.2^2) ≈ 4.00213469955E-2.`

`[.5 .255700574089]`, the exact value being
`tan(.5^2)  ≈ .255341921221.`

Then press "EXIT" to quit the program.

PARTIAL SUMS OF A FOURIER SERIES

Let f be a periodic function with period T > 0. We assume that the expression of f is known over the interval [a, b], where b=a+T.
The Fourier series of f, as long as it converges, is written:

$$\sum_{k=0}^{+\infty} \left(ak*\cos(k*w*x) + bk*\sin(k*w*x) \right) \quad \text{where} \quad w = \frac{2*\pi}{T} .$$

where, for all values of k ≥ 0,

$$ak = \frac{2}{T} \int_a^b f(x)*\cos(k*w*x)dx \quad \text{and} \quad bk = \frac{2}{T} \int_a^b f(x)*\sin(k*w*x)dx$$

(Exception:

$$a0 = \frac{1}{T} \int_a^b f(x)dx)$$

'FOUR' allows you to calculate any partial sum in this series, i.e. any expression like:

$$\sum_{k=m}^{n} \left(ak*\cos(k*w*x) + bk*\sin(k*w*x) \right) .$$

The functional diagram is as follows:

	Before			After
3:	list		3:	
2:	m	'FOUR'	2:	
1:	n	——>	1:	partial sum

where "list" is a list in the format { F a b } where:
F is the expression of the function (a capital X must be used)
a and b are the end points of the interval (the period is taken to be equal to T = b-a)
m and n are the lower and upper limit k respectively for which we calculate ak and bk.

Integrals are calculated to the degree of accuracy set by the display mode (in **n FIX** mode, integrals are correct to n decimal places).

The result or "partial sum" is obtained in the form of an algebraic expression in which w and x are denoted symbolically by the variables 'X' and 'W'. By pressing EVAL we can obtain the value of 'W', i.e.:

$$W = \frac{2\pi}{T} \quad \text{(W="angular frequency").}$$

TEXT OF PROGRAM 'FOUR'

'FOUR': (Checksum: # 37251d, Size: 380.5 bytes)

```
«  →  m  n
   «  'X'  PURGE  EVAL  DUP2  -  NEG
      →  f  a  b  t
      «  RAD  -2  SF  2  π  *  t  /  'W'  STO
         «  a  b  ROT  'X'  ∫  →NUM  DUP
            ABS  IERR  >  *  t  /
         »
         →  int
         «  0  m  n  FOR  k
               IF  k  THEN
                  k  'W'  *  'X'  *
                  →  h
                  «  h  EVAL  COS  f  *  int  EVAL  2  *
                     h  COS  *  +  h  EVAL  SIN  f  *
                     int  EVAL  2  *  h  SIN  *  +
                  »
               ELSE  f  int  EVAL  +  END
            NEXT
         »
      »
   »
»
```

Note:

If, during computation, program 'FOUR' finds a value for an integral with an absolute value that is less than the absolute error given by the calculator for that integral, it considers it to be zero and the corresponding quantity does not appear in the expression of the partial sum of the Fourier series.

This is often the case with even functions (where values of "bk" are zero) or odd functions (where values of "ak" are zero).

In such cases, any negligible terms are removed from the expression of the partial sum of the Fourier series.

PROGRAM 'FOUR': PRACTICAL EXAMPLES

Example 1:

Expand the periodic function f, with period 1, equal to X^2 over the closed interval `[ 0, 1 ]` into a Fourier series. We want, for example, to calculate the coefficients ak and bk for `0 ≤ k ≤ 3`. Integrals are calculated in 3 FIX mode. This gives us the following, within 33 seconds:

3:	{ 'X*X' 0 1 }		3:	
2:	0	'FOUR' ——>	2:	
1:	3		1:	expression

where expression =

```
'0.333+0.101*COS(W*X)-0.318*SIN(W*X)+0.025*COS(2*W*X)
  -0.159*SIN(2*W*X)+0.011*COS(3*W*X)-0.106*SIN(3*W*X)'
```

By pressing EVAL we can then replace 'W' with its numerical value, i.e. `2*π/T = 2*π ≈ 6.283`.
In this example, we can concentrate solely on the coefficients a5 and b5, simply by entering 5 and 5 at levels 2 and 1.
We then obtain within 16 seconds at level 1 of the stack, the expression:

```
'0.004*COS(5*W*X)-0.064*SIN(5*W*X).
```

Example 2:

Let us now look at the same function `X --> X²`, but expanding over the interval `[ 0, 1 ]` into a cosine series. We simply have to take the function to be even and defined over `[ 1, -1 ]`, and therefore periodic with period 2.

In 3 FIX mode, we obtain within 26 seconds:

3:	{ 'X*X' -1 1 }		3:	
2:	0	'FOUR' ——>	2:	
1:	3		1:	expression

where the expression =

```
'0.333-0.405*COS(W*X)+0.101*COS(2*W*X)-0.045*COS(3*W*X)'
```

We can then use EVAL to replace 'W' with its numerical value `(2*π/T=π≈3.142)`.

Example 3:

We can also expand X^2 over `[0,1]` into a sine series. We simply have to extend evenly over `[-1,0]` and take the period to be 2. We enter `{ 'X*X*SIGN(X)' -1 1 }` at level 3 (other levels remaining the same as in example 2). We then obtain (in 3 FIX mode):

We can then use EVAL to replace 'W' with its numerical value `( 2π/T = π )`.

FINITE SERIES

The HP48 is capable of calculating a finite series at 0 of degree n of a function f, using Taylor's formula (provided that f is sufficiently differentiable):

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots + \frac{f^{(n)}(0)}{n!}x + o(x^n).$$

To arrive at a result in this way, we have to use the TAYLR instruction in the ALGBRA menu. The problem with this method is that the HP48 calculates the successive derivatives of f, in symbolic form, before evaluating them for x = 0. Computation is sometimes extremely lengthy and takes up large amounts of memory.

To take an example, the finite series at zero of degree 5 of the function f defined by `f(x) = exp(sin(x))` (hardly a complicated example!) is obtained in 1 min. 30 s, giving the solution:

```
'1+X+.5*X^2-.125*X^4-6.66666666667E-2*X^5'.
```

Using the same data, the programs in the 'FS' directory can find the same result in 11 seconds.
Other more convincing examples are not lacking.
If we want to calculate the series of `ATAN(ASIN(X))` (arc tangent of arc sine of x), at 0 of degree 5, using the TAYLR instruction in the ALGBRA menu, computation is halted after 2 minutes and an "Insufficient Memory" error message is displayed (even though the calculator still has 4,500 bytes of free memory). On the same calculator, the same series can be obtained in 11 seconds with the programs in the 'FS' directory, giving the solution:

```
'X-.166666666666*X^3+.108333333333*X^5'
```

The time thus saved is considerable, and valuable memory space essential to computation is also saved.

The main idea behind the programs in the 'FS' directory is that a finite series like $f(x) = a + bx + cx^2 + \ldots + dx^n + o(x^n)$ can be represented by the vector: `[ a b c .... d ]`.

It is in this form (in increasing order of powers of x) that finite series are used and obtained in the 'FS' directory.
To make series easier to read, program 'FS→' transforms such vectors by expressing them algebraically.
Each of the main operations performed on finite series (sum, product, power, quotient, composite) required a separate program, and calculation of the finite series of each standard function had to be programmed. The directory therefore contains a specific program for each standard function (16 here), which leaves us with 27 programs in total.
We could have reduced the number of programs by storing standard series with a degree of up to 7, for example, in a matrix (this would have given us an array of `7*16=112` real numbers occupying `112*6=672` bytes of memory). But this posed a dual problem: taking the same example, the degree of any finite series would have to be less than or equal to 7 and it would therefore have been impossible to take advantage of the properties of the various functions.

Important note: Some programs in the 'FS' directory call programs in the 'POLY' directory.

Here is the list of programs in the 'FS' directory:

Operations on finite series:

'DIM' : Degree and index of the first non-zero term of a given finite series. ('DIM' is used very regularly)
'ΣFS' : sum of two finite series.
'πFS' : product of two finite series.
'CPFS' : composite of two finite series.
'QFS' : quotient of two finite series.
'FSN' : integer power of a finite series.

Standard operations on finite series:

'FS→' : Changes from "vector" form to conventional algebraic form of a finite series.
'DERFS' : derivative of a finite series.
'INTFS' : integration of a finite series.
'X→XN' : replacing X with X^n in a finite series.
'X→AX' : replacing X with a*X in a finite series.

Standard finite series and composition using standard finite series:

'EX', 'AX', 'LG', 'XA';
'SN', 'CS', 'TG', 'SH', 'CH', 'TH';
'ASNS', 'ACS', 'ATG', 'ASH', 'ATH';

These programs allow you to calculate the finite series of each of the functions f shown below, and to calculate the composite of a given finite series using such a function f. These functions f, in the order given above, are:

```
X --> EXP(X), X --> A^X, X --> LN(X), X --> X^A;
X --> SIN(X), X --> COS(X), X --> TAN(X);
X --> SINH(X), X --> COSH(X), X --> TANH(X);
X --> ASIN(X), X --> ACOS(X), X --> ATAN(X);
X --> ASINH(X), X --> ATANH(X);
```

N.B:

The programs in the 'FS' directory accept finite series as arguments and give results in the form of finite series.
When used as an argument, a finite series must be given in vector form. The programs in the 'FS' directory will interpret the length of the vector as giving the degree of the finite series. For example, if you put `[ 1 3 -4 ]` into the stack, the corresponding finite series will be `1 + 3*X -4*X² + o(X²)`. To calculate the finite series of `SIN(1 + 3*X -4*X²)` of degree 6, you will need to put `[ 1 3 -4 0 0 0 0 ]` into the stack (corresponding to the finite series `1 + 3*X -4*X² + o(X^6))`.
Likewise, when a program in the 'FS' directory gives a finite series as a result, the series is calculated to the largest possible degree (depending on the data entered, of course) and all the coefficients obtained are exact (neglecting round-off errors, which are often negligible). All the finite series looked at here will be taken to be finite series at zero (this is an absolute requirement for the F.S. of g when determining the F.S. of gof).

DEGREE AND INDEX OF THE FIRST NON-ZERO TERM OF A FINITE SERIES

If `f(x) = A0 + A1*X + A2*X² + ... + An*Xⁿ + o(Xⁿ)` is a finite series, n is its degree and p is the smallest index such that $Ap \neq 0$. p is equal to zero if the constant term is non-zero.
n and p are very important in operations on finite series (product, quotient, composite), as they allow us to calculate the degree of the finite series obtained from such operations.

'DIM' calculates these two indices from a finite series expressed in vector form.
The functional diagram of the program is as follows:

2:		'DIM'	2:	degree
1:	Vector	————>	1:	index p

Here the degree n is equal to the dimension of the vector (given by SIZE) minus 1. A finite series of degree n is in fact represented by a vector of n+1 coefficients.
The index p is an integer such that $0 \leq p \leq n$. An exception to this rule arises when the vector at level 1 contains zero coefficients only, in which case we get $p=n+1$.

Note: the user should not have to call program 'DIM' himself, as it is called by most of the programs in the 'FS' directory.

'DIM': (Checksum: # 9826d, Size: 105 bytes)

```
«   DUP   SIZE   1   GET   1   -   SWAP
    IF   DUP   RNRM
         THEN
              1   DO   GETI   UNTIL   ABS   END
              SWAP   DROP   2   -   OVER   1   +   MOD
         ELSE
              DROP   DUP   1   +
    END
»
```

Examples:

2:		'DIM'	2:	3
1:	[1 0 5 8]	————>	1:	0

2:		'DIM'	2	4
1:	[0 0 2 3 0]	————>	1:	2

2:		'DIM'	2:	2
1:	[0 0 0]	————>	1:	3

WRITING A FINITE SERIES IN ALGEBRAIC FORM

'FS→' expresses a finite series written in vector form algebraically. The variable used is X.

The last term of the algebraic expression of the series is in the form `o(X^n)`, which indicates the degree. This makes it possible to distinguish between finite series like, for example, `[ 1 -2 5 ]` and `[ 1 -2 5 0 0 ]`, which would otherwise give the same expression.

The functional diagram is as follows:

```
1: [ vector ]   --'FS→'-->   1: [ expression ]
```

'FS→':(Checksum: # 31065d, Size: 156.5 bytes)

```
«   OBJ→   1   GET   1   -
    →   n
    «   0
        0   n   FOR   i
            n   i   -   2   +   ROLL   'X'   i   ^   *   +
        NEXT
        'o(X^Y)'   {Y}   n   +   ↓MATCH   DROP   +
    »
»
```

Examples: (computation time = 3 seconds)

```
1: [ 1 -2 5 ]             --'FS→'-->   1: '1-2*X+5*X^2+o(X^2)'

1: [ 1 -2 5 0 0 ]         --'FS→'-->   1: '1-2*X+5*X^2+o(X^4)'

1: [ 0 0 0 0 0 0 ]        --'FS→'-->   1: 'o(X^5)'
```

SUM OF TWO FINITE SERIES

'ΣFS' calculates the sum of two finite series FS1 and FS2, both expressed in vector form. The result obtained is thus a vector.
If the two series have the same degree, adding them is like calculating the sum of two vectors. Otherwise, the sum is calculated after truncating the series with the highest degree.
If FS1 and FS2 are of degree n and p, the result obtained is a finite series of minimum degree (n,p).

The functional diagram is as follows:

```
2:  FS2          'ΣFS'      2:
                 ------>
1:  FS1                     1:  FS1 + FS2
```

'ΣFS':(Checksum: # 26598d, Size: 62.5 bytes)

```
«   DUP2
    SIZE  1  GET  SWAP  SIZE  1  GET
    MIN  1  →LIST  SWAP
    OVER  RDM  3  ROLLD  RDM  +
»
```

Examples: (in less than one second)

```
2:  [ 1 5 -2 3 ]           'ΣFS'      2:
                           ------>
1:  [ 4 3  2 2 ]                      1:  [ 5 8 0 5 ]

2:  [ 1 5 -2 3 ]           'ΣFS'      2:
                           ------>
1:  [ 4 3  2 2 1 9 2 ]                1:  [ 5 8 0 5 ]

2:  [ 1 5 -2 3 ]           'ΣFS'      2:
                           ------>
1:  [ 0 0 0 ]                         1:  [ 1 5 -2 ]
```

PRODUCT OF TWO FINITE SERIES

'πFS' calculates the product of two finite series FS1 and FS2, both expressed in vector form. The result obtained is thus a vector.
The degree of the finite series depends on the degree and the index of the first non-zero term of the two series FS1 and FS2. 'πFS' always gives the result with the highest degree (all coefficients obtained are exact).

The functional diagram is as follows:

```
2: |   FS2   |     'πFS'      2: |         |
               ------------>
1: |   FS1   |                1: | FS1*FS2 |
```

N.B: program 'πFS' calls 'DIM' and program 'PRODP' in the 'POLY' directory.

'πFS':(Checksum: # 3903d, Size: 94.5 bytes)

```
«  DUP2  DIM  ROT  DIM  4  ROLL  +  3  ROLLD
   +  MIN  1  +  1  →LIST  3  ROLLD
   POLY  PRODP  FS  SWAP  RDM
»
```

Examples:

* in less than 2 seconds:

```
2: | [ 1 -2 3 0 1 ] |    'πFS'    2: |                   |
                      ---------->
1: | [ 2 0 1 -1 3 ] |             1: | [ 2 -4  7 -3 10 ] |
```

```
As ( 1 -2*X +3*X^2 +X^4 +o(X^4) ) * ( 2 +X^2 -X^3 +3*X^4 +o(X^4) )
       = 2 -4*X +7*X^2 -3*X^3 +10X^4 +o(X^4).
```

* in 2 seconds:

```
2: | [ 0 0 1 3 2 0 ] |    'πFS'    2: |                        |
                       ---------->
1: | [ 0 -1 5 0 0 ]  |             1: | [ 0 0 0 -1 2 13 10 ]   |
```

```
As ( X^2 +3*X^3 +2*X^4 +o(X^5) ) * ( -X +5*X^2 +o(X^4) )
   = -X^3 +2*X^4 +13*X^5 +10*X^6 +o(X^6)
```

COMPOSITE OF TWO FINITE SERIES

'CPFS' calculates the composite of two finite series FS1 and FS2, both expressed in vector form. The result obtained is thus a vector.
The degree of the finite series obtained depends on the degree and the index of the first non-zero term of each of the two series FS1 and FS2. 'CPFS' always gives the result with the highest degree (all coefficients obtained are exact).

The functional diagram is as follows:

2:	FS2	'CPFS'	2:	
1:	FS1	————>	1:	FS1 o FS2

N.B: program 'CPFS' calls 'DIM' and program 'PRODP' in the 'POLY' directory.
It is essential that the constant term (the value of FS2 at zero) be zero. If not, an error message is displayed.

'CPFS':(Checksum: # 23153d, Size: 326 bytes)

```
«  IF    OVER  1  GET  ABS  NOT  THEN
         DUP  1  0  PUT  SWAP  {1}  RDM
         3  PICK  DIM  4  PICK  DIM
         →  sf  vf  sg  vg
         «  IF  sg  THEN
                'vf*(vg-1)+sf+1'  EVAL  'vf*(sg+1)'  EVAL
                MIN  1  →LIST
                →  d
                «  d  RDM
                   POLY  1  1  →ARRY
                   2  sg  1  +  FOR  i
                      4  PICK  PRODP  d  RDM
                      DUP  4  PICK  i  GET  *
                      ROT  +  SWAP
                   NEXT
                   DROP  FS
                »
            END  3  ROLLD  DROP2
         »
   END
»
```

Example: in 4 seconds,

2:	[0 0 -1 2 1]	'CPFS'	2:	
1:	[1 0 -2 1]	————>	1:	[1 0 0 0 -2 8 -5]

Since if $\quad$ `f(X) = -X^2 +2*X^3 +X^4 +o(X^4)`
and if $\quad$ `g(X) = 1 -2*X^2 +X^3 +o(X^3)`, then:
`gof(X) = 1 -2*X^4 +8*X^5 -5*X^6 +o(X^6).`

INVERSE OF A FINITE SERIES AND SERIES OF 1/(1 + X)

'INVFS' finds the finite series of `1/(1+X)` or `1/(1-X)` of degree n, and also calculates the inverse of a finite series FS1. (The result obtained is also a finite series, in which case it is essential for the constant term of FS1 to be non-zero, otherwise a "divide by zero" error will occur).

The functional diagram is as follows:

```
1: | n ( ≥ 0 ) |   'INVFS'  ──>  1: | FS of 1/(1+X) of degree n |
or:
1: | n ( < 0 ) |   'INVFS'  ──>  1: | FS of 1/(1-X) of degree (-n) |
or:
1: | FS1 |         'INVFS'  ──>  1: | FS of 1/FS1 |
```

'INVFS':(Checksum: # 46632d, Size: 192.5 bytes)

```
«  «   →  n  s
       «  0  n  FOR  i  s  i  ^  NEXT  n  1  +  →ARRY
       »
   »
   SWAP  DUP
   IF  DUP  TYPE  THEN
       1  GET  DUP  INV  3  ROLLD  /  1  0  PUT
       DUP  DIM  /  FLOOR  -1  5  ROLL  EVAL  CPFS  *
   ELSE
       ABS  SWAP  SIGN  NEG  ROT  EVAL
   END
»
```

Example:

* FS of `1/(1+X)` of degree 5: (in under one second)

```
1: | 5 |   'INVFS'  ──>  1: | [ 1 -1 1 -1 1 -1 ] |
```

* FS of `1/(1-X)` of degree 7: (in under one second)

```
1: | -7 |   'INVFS'  ──>  1: | [ 1 1 1 1 1 1 1 1 ] |
```

FS of `1/(1 -X^2 + X^3 -2*X^4 + o(X^5))`: in 3 seconds,

```
1: | [ 1 0 -1 1 -2 0 ] |   'INVFS'  ──>  1: | [ 1 0 1 -1 3 -2 ] |
```

which gives:

```
1/( 1 -X^2 + X^3 -2*X^4 + o(X^5)) = 1 +X^2 -X^3 +3*X^4 -2*X^5 +o(X^5)
```

QUOTIENT OF TWO FINITE SERIES

'QFS' calculates the quotient of two finite series FS1 and FS2, both expressed in vector form. The result obtained is thus a vector.
The degree of the finite series obtained depends on the degree and the index of the first non-zero term of the two series FS1 and FS2. 'QFS' always gives the result with the highest possible degree (all coefficients obtained are exact).

The functional diagram is as follows:

```
2: | FS2 |      'QFS'      2: |           |
                ------->
1: | FS1 |                 1: | FS2 / FS1 |
```

N.B: program 'QFS' calls 'DIM' and program 'DIVIP' in the 'POLY' directory.
The index p of the first non-zero term of FS2 or FS1 may be zero. It is essential, however, that the index p of FS1 be less than or equal to that of FS2. If not, an error message will be displayed.

'QFS':(Checksum: # 2650d, Size: 190.5 bytes)

```
«   DUP2   DIM    ROT    DIM    ROT    MIN    3    ROLLD
    MIN    OVER   -    1    +    1    →LIST
    →    q    r
    «    «    OBJ→    1    GET    q    -    →ARRY
              q    1    +    ROLLD    q    DROPN    r    RDM
         »
         ROT    OVER    EVAL    3    ROLLD    EVAL    r    1    GET
         POLY   DIVIP   FS
         DROP   r    RDM
    »
»
```

Example: in under four seconds:

```
2: | [ 0 0 2 3 1 -1 ] |    'QFS'    2: |                |
                          ------->
1: | [ 0 1 -2 1 1 ]   |             1: | [ 0 2 7 13 ]   |
```

Which gives:

if $\quad$ f(X) = 2*X^2 + 3*X^3 +X^4 -X^5 +o(X^5)
and if $\quad$ g(X) = X -2*X^2 +X^3 +X^4 +o(X^4), then:

$$\frac{f(X)}{g(X)} = 2*X +7*X^2 +13*X^3 + o(X^3).$$

RAISING A FINITE SERIES TO AN INTEGER POWER

'FSN' allows you to raise a finite series FS1 (written in vector form) to an integer power n (where n is zero or a positive integer). The result obtained is a vector representing $FS1^n$.

The degree of the series obtained depends on the degree and the index of the first non-zero term of FS1. 'FSN' always gives the result to the highest possible degree (all coefficients obtained are exact).

The functional diagram is as follows:

2:	FS1	'FSN'	2:	
1:	n	——>	1:	$FS1^n$

N.B: 'FSN' calls 'πFS'. 'FSN' also calls itself.

'FSN':(Checksum: # 50525d, Size: 104.5 bytes)

```
«   IF    DUP    1    ≤    THEN    DROP
    ELSE
          OVER    DUP    πFS    OVER    2    /    FLOOR    FSN
          IF    SWAP    2    MOD
                THEN    πFS
                ELSE    SWAP    DROP
          END
    END
»
```

Examples:

* in 4 seconds:

2:	[1 1 2 -1]	'FSN'	2:	
1:	5	——>	1:	[1 5 20 45]

as (1 +X +2*X^2 -X^3 +o(X^3))^5 = 1 +5*X +20*X^2 +45*X^3 +o(X^3).

* in under 4 seconds:

2:	[0 1 -1 2 1 0 0]	'FSN'	2:	
1:	4	——>	1:	[0 0 0 0 1 -4 14 -24 37 -20]

as (X -X^2+2*X^3+X^4+o(X^6))^4
= X^4 -4*X^5 +14*X^6 -24*X^7 +37*X^8 -20*X^9 +o(X^9).

TRANSFORMATION X → X^N IN A FINITE SERIES

'X→XN' allows you to transform the finite series of F(X) into a finite series of $F(X^n)$, where n is a natural integer.

The functional diagram is as follows:

```
2: | FS of F(X) |   'X→XN'   2: |              |
1: |     n      |  ------->  1: | FS of F(X^n) |
```

'X→XN':(Checksum: # 4914d, Size: 48.5 bytes)

```
«   OVER   SIZE   +      RDM
    TRN    DUP    SIZE   EVAL   *
    1      →LIST  RDM
»
```

Example: (in under one second)

```
2: | [ 1 3 4 -1 ] |   'X→XN'   2: |                                |
1: |      3       |  ------->  1: | [ 1 0 0 3 0 0 4 0 0 -1 0 0 ] |
```

Since, if `F(X)= 1 +3*X +4*X^2 -X^3 +o(X^4)`, then
`F(X^3)= 1 +3*X^3 +4*X^6 -X^9 +o(X^11).`

TRANSFORMATION X → A*X IN A FINITE SERIES

'X→AX' allows you to transform the finite series of F(X) into a finite series of F(aX), where a is a scalar.

The functional diagram is as follows:

2:	FS of F(X)	'X→AX'	2:	
1:	a	——>	1:	FS of F(aX)

<u>N.B</u>: program 'X→AX' calls program 'DIM'.

'X→AX':(Checksum: # 55099d, Size: 78.5 bytes)

```
«   SWAP   DUP   DIM   FOR   i
         i    1   +   DUP2   GET
         4   PICK   i   ^   *   PUT
    -1   STEP
    SWAP   DROP
»
```

<u>Example</u>: (in under one second)

2:	[1 2 -1 3 1]	'X→AX'	2:	
1:	-2	——>	1:	[1 -4 -4 -24 16]

Since, if F(X) = 1 +2*X -X^2 +3*X^3 +X^4 +o(X^4),
F(-2X)= 1 -4*X -4*X^2 -24*X^3 +16*X^4 +o(X^4).

DERIVATIVE OF A FINITE SERIES

'DERFS' allows you to obtain a finite series `F'(X)` from a finite series F(X) by deriving each coefficient term by term (provided that F is sufficiently differentiable.).

The functional diagram is as follows:

```
          'DERFS'
1: FS of F(X)   ------->   1: FS of F'(X)
```

The degree of the finite series obtained is obviously one less than the degree of the initial series.

'DERFS':(Checksum: # 29761d, Size: 121.5 bytes)

```
« OBJ→  1  GET  1  -
  IF  DUP  THEN
       →  t
       «  t  1  FOR  i  i  *  t  ROLLD  -1  STEP
          t  →ARRY  SWAP  DROP
       »
  ELSE  DROP2  0  1  →ARRY  END
»
```

Example: (in under one second)

```
                              'DERFS'
1: [ 2 -1 3 2 8 5 1 ]   ------->   1: [ -1 6 6 32 25 6 ]
```

The finite series:

`2 -X +3*X^2 +2*X^3 +8*X^4 +5*X^5 +X^6 +o(X^6)`

therefore becomes:

`-1 +6*X +6*X^2 +32*X^3 +25*X^4 +6*X^5 +o(X^5).`

INTEGRATING A FINITE SERIES

'INTFS' allows you to obtain the primitive G(X), cancelling to zero, of the finite series F(X) by integrating each coefficient term by term.

The functional diagram is as follows:

```
1: [ FS of F(X) ]   'INTFS'
                   -------->   1: [ FS of G(X) ]
```

The degree of the finite series obtained is obviously one more than the degree of the initial series. Its constant term (i.e. the first coefficient of the vector obtained) is zero.

'INTFS':(Checksum: # 64552d, Size: 89 bytes)

```
«   0    SWAP   OBJ→   1   GET
    →    t
    «    t    1    FOR   i
              i    /     t    ROLLD
         -1   STEP
         t    1    +     →ARRY
    »
»
```

Example: (in under one second)

```
1: [ 2 -1 3 2 8 ]   'INTFS'
                   -------->   1: [ 0 2 -.5 1 .5 1.6]
```

The finite series:

```
2 -X +3*X^2 +2*X^3 +8*X^4 +o(X^4)
```

therefore becomes:

```
2*X -.5*X^2 +X^3 +.5*X^4 +1.6*X^5 +o(X^5).
```

FINITE SERIES OF EXP(X) AND EXPONENTIAL FUNCTION OF AN F.S.

'EX' lets you calculate the finite series of `EXP(X)` or `EXP(-X)` of degree n, or the finite series of `EXP(F(X))` where F is entirely expressed by its finite series.
The functional diagram is as follows:

```
1: | n ( ≥ 0 )   |   'EX'  ---->  1: | FS of EXP(X) of degree n      |
or:
1: | n ( < 0 )   |   'EX'  ---->  1: | FS of EXP(-X) of degree (-n)  |
or:
1: | FS of F(X)  |   'EX'  ---->  1: | FS of EXP(F(X))               |
```

In the latter case, the F.S. of `EXP(F(X))` is obtained with the same degree as that of F(X).
N.B: 'EX' calls programs 'DIM' and 'CPFS'.

'EX':(Checksum: # 38699d, Size: 189 bytes)

```
«  «   →  n   s
       «  0   n   FOR  i  s  i  ^  i  !  /  NEXT
          n   1   +   →ARRY
       »
   »
   SWAP   DUP
   IF   DUP   TYPE   THEN
        1   GET   EXP   SWAP   1   0   PUT   DUP
        DIM   /   FLOOR   1   5   ROLL   EVAL   CPFS   *
   ELSE
        ABS   SWAP   SIGN   ROT   EVAL
   END
»
```

Examples:

* in under one second:

```
1: |  3  |   'EX'  ---->  1: | [ 1  1  .5  .166666666667 ] |
```

* in under one second:

```
1: | -4  |   'EX'  ---->  1: | [1 -1 .5 -.166666666667 4.16666666667E-2] |
```

* in 7 seconds:

```
| [0 6 -1 2 1 5] |  'EX'  ---->1: | [ 1 6 17 32.0000000001 49.5 76.8000000001 ] |
```

as: `EXP( 6*X -X^2 +2*X^3 +X^4 +5*X^5 + o(X^5) )`
`= 1 +6*X +17*X^2 +32*X^3 +49.5*X^4 +76.8*X^5 +o(X^5)`

FINITE SERIES OF A^X AND F.S. OF A^F(X)

'AX' lets you calculate the finite series of `A^X` or `A^(-X)` of degree n, or the finite series of `A^(F(X))` where F is entirely expressed by its finite series. A must be > 0.
The functional diagram is as follows:

```
2:  |        A         |    'AX'    2: |                       |
1:  |    n  ( ≥ 0 )    |   ------>  1: | FS of A^X) of degree n |
```

or:

```
2:  |        A         |    'AX'    2: |                            |
1:  |    n  ( < 0 )    |   ------>  1: | FS of A^(-X) of degree (-n) |
```

or:

```
2:  |        A         |    'AX'    2: |                  |
1:  |   FS  of F(X)    |   ------>  1: |   FS of A^F(X)   |
```

In the latter case, the F.S. of `A^(F(X))` is obtained with the same degree as that of `F(X)`.
N.B: 'AX' calls programs 'EX' and 'X→AX'.

'AX':(Checksum: # 32770d, Size: 72.5 bytes)

```
«   IF    DUP   TYPE   THEN   SWAP   LN    *   EX
    ELSE  EX    SWAP   LN     X→AX   END
»
```

Examples:

* in one second:

```
|  2  |   'AX'      |                                                        |
|  3  |  ------> 1: | [1 .69314718056 .240226506959 5.55041086649E-2 ]       |
```

As:

`2^X= 1+.69314718056*X .240226506959*X^2 +5.55041086649E-2*X^3 +o(X^3).`

* in one second:

```
|  2  |             |                                                        |
| -3  |  ------> 1: | [1 -.69314718056 .240226506959 -5.55041086649E-2 ]     |
```

(we thus obtain the F.S. of `2^(-X)` of degree 3).

* in under 4 seconds:

```
2: |       5        |   'AX'     |  [ 4.99999999998   0   8.04718956212   |
1: | [1 0 1 3 -2 ]  |  ------> 1:|    24.1415686864   -9.61865313931 ]   |
```

We thus obtain the F.S. of `5^(1 +X^2 +3*X^3 -2*X^4 +o(X^4))` of degree 4.

FINITE SERIES OF LN(1+X) AND NAPIERIAN LOG OF AN F.S.

'LG' lets you calculate the finite series of `LN(1+X)` or `LN(1-X)` of degree n, or the finite series of `LN(F(X))` where F is entirely expressed by its finite series.
The functional diagram is as follows:

```
1: [ n ( ≥ 0 ) ]   'LG' ---->   1: [ FS of LN(1+X) of degree n ]
or:
1: [ n ( < 0 ) ]   'LG' ---->   1: [ FS of LN(1-X) of degree (-n) ]
or:
1: [ FS of F(X) ]  'LG' ---->   1: [ FS of LN(F(X)) ]
```

In the latter case, the F.S. of `LN(F(X))` is obtained with the same degree as that of `F(X)`.
We should also obtain `F(0) > 0`.
N.B: 'LG' calls programs 'DIM' and 'CPFS'.

'LG':(Checksum: # 54237d, Size: 223.5 bytes)

```
«  «  →  n  s
      «  0  IF  n  THEN  1  n  FOR  i  s  i  ^  i  /  NEXT END
         n  1  +  →ARRY  NEG
      »
   »
   SWAP  DUP
   IF  DUP  TYPE  THEN
      1  GET  DUP  LN  3  ROLLD  /  1  0  PUT  DUP
      DIM  /  FLOOR  -1  5  ROLL  EVAL  CPFS  1  ROT  PUT
   ELSE
      ABS  SWAP  SIGN  NEG  ROT  EVAL
   END
»
```

Examples:
* in under one second:

```
1: [ -3 ]   'LG' ---->  1: [ [ 0  -1  -.5  -.333333333333 ] ]
```

* in under one second:

```
1: [ 4 ]   'LG' ---->  1: [ [ 0  1  -.5  .333333333333  -.25 ] ]
```

* in 5 seconds:

```
[2 1 -3 4 1] ->1: [.69314718056 .5 -1.625 2.79166666667 -2.015625]
```

as `LN( 2 +X -3*X^2 +4*X^3 +X^4 +o(X^4))=`
`.69314718056+.5*X-1.625 *X^2+2.79166666667*X^3-2.015625*X^4 +o(X^4).`

FINITE SERIES OF (1+X)^A AND F.S. OF F(X)^A

'XA' lets you calculate the finite series of `(1+X)^A` or `(1-X)^A` of degree n, or the finite series of `(F(X))^A` where F is entirely expressed by its finite series.
The functional diagram is as follows:

```
      +----------------+          +----------------------------+
2:    |   n ( ≥ 0 )    |   'XA'  2:|                            |
      +----------------+  ------> +----------------------------+
1:    |       A        |         1:| FS of (1+X)^A of degree n  |
      +----------------+          +----------------------------+
```

or:

```
      +----------------+          +----------------------------------+
2:    |   n ( < 0 )    |   'XA'  2:|                                  |
      +----------------+  ------> +----------------------------------+
1:    |       A        |         1:| FS of (1-X)^A of degree (-n)     |
      +----------------+          +----------------------------------+
```

or:

```
      +----------------+          +---------------------------+
2:    |   FS of F(X)   |   'XA'  2:|                           |
      +----------------+  ------> +---------------------------+
1:    |       A        |         1:|       FS of F(X)^A        |
      +----------------+          +---------------------------+
```

In the latter case, the F.S. of `F(X)^A` is obtained with the same degree as that of F(X). F(0) must be positive.
N.B: 'XA' calls programs 'DIM' and 'CPFS'.

'XA': (Checksum: # 11288d, Size: 254.5 bytes)

```
« → a
  « « → n s
        « 1 IF n THEN 1 n FOR i DUP 's*(a-i+1)/i'
          EVAL * NEXT END n 1 + →ARRY
        »
    »
    SWAP DUP
    IF DUP TYPE THEN
        1 GET DUP a ^ 3 ROLLD / 1 0 PUT
        DUP DIM / FLOOR 1 5 ROLL EVAL CPFS *
    ELSE ABS SWAP SIGN ROT EVAL END
  »
»
```

Example: (in under one second)

```
      +----------------+          +-----------------------------------+
2:    |       4        |   'XA'  2:|                                   |
      +----------------+  ------> +-----------------------------------+
1:    |      .5        |         1:| [1 .5 -.125 .0625 -.0390625]      |
      +----------------+          +-----------------------------------+
```

as `(1+X)^(1/2)=1+(1/2)*X-(1/8)*X^2+(1/16)*X^3-(5/128)*X^4 +o(X^4)`

Note: If you are likely to run out of memory on your HP48, 'XA' can be written:

```
« SWAP LG * EX »
```

This is obviously a lot shorter, but is sometimes a little less accurate (as shown in the example above) and is certainly slower.

FINITE SERIES OF SIN(X) AND SINE OF AN F.S.

'SN' lets you calculate the finite series of `SIN(X)` of degree n `(n ≥ 0)`, or the finite series of `SIN(F(X)` where F is entirely expressed by its finite series.
The functional diagram is as follows:

```
                          'SN'
1: |     n ( ≥ 0 )     |  ----->  1: | FS of SIN(X) of degree n |
```

or:

```
                          'SN'
1: |    FS of F(X)     |  ----->  1: | FS of SIN(F(X)) |
```

In the latter case, the F.S. of `SIN(F(X)` is obtained with the same degree as that of `F(X)`.
N.B: 'SN' calls programs 'EX' and 'X→AX'.

'SN':(Checksum: # 2996d, Size: 123 bytes)

```
«   IF   DUP    TYPE    THEN
              (0,1)   *    EX
         ELSE
              EX    (1,0)    *    (0,1)    X→AX
    END
    IM
»
```

Examples:

* in two seconds:

```
               'SN'
1: |    4    |  ----->  1: | [ 0  1  0  -.166666666667  0 ] |
```

* in 7 seconds:

```
                      'SN'
1: | [0 1 2 -1 3] |  ----->  1: | [  0  1  2  -1.16666666667  2  ] |
```

As `SIN( X +2*X^2 -X^3 +3*X^4 +o(X^4) )`
`= X +2*X^2 -1.16666666667*X^3 +2*X^4 +o(X^4).`

FINITE SERIES OF COS(X) AND COSINE OF AN F.S.

'CS' lets you calculate the finite series of `COS(X)` of degree n (`n ≥ 0`), or the finite series of `COS(F(X)` where F is entirely expressed by its finite series.
The functional diagram is as follows:

```
                            'CS'
1: | n ( ≥ 0 )      |      ------>  1: | FS of COS(X) of degree n |
```

or:

```
                            'CS'
1: | FS of F(X)     |      ------>  1: | FS of COS(F(X)) |
```

In the latter case, the F.S. of `COS(F(X)` is obtained to the maximum possible degree, given the index p of the F.S. of `F(X)`.
N.B: 'CS' calls programs 'DIM' 'CPFS', 'EX' and 'X→AX'.

'CS':(Checksum: # 7756d, Size: 138 bytes)

```
«  «  EX   (1,0)   *   (0,1)   XîAX   RE
   »
   IF   OVER   TYPE   THEN
        OVER   DIM   /   FLOOR   1   +   SWAP   EVAL   CPFS
   ELSE   EVAL
   END
»
```

Examples:

* in two seconds:

```
                 'CS'
1: |    5    |  ------> 1: | [ 1  0 -.5  0 4.16666666667E-2  0 ] |
```

* in 6 seconds:

```
                        'CS'
1: | [0 0 1 -3 1 5] |  ------> 1: | [  1  0  0  0  -.5  3  -5.5  -2 ] |
```

As `COS( X^2 -3*X^3 +X^4 +5*X^5 +o(X^5) )`
`=   1 -.5*X^4 +3*X^5 -5.5*X^6 -2*X^7 +o(X^7).`

FINITE SERIES OF TAN(X) AND TANGENT OF AN F.S.

'TG' lets you calculate the finite series of TAN(X) of degree n (n ≥ 0), or the finite series of `TAN(F(X)` where F is entirely expressed by its finite series.
The functional diagram is as follows:

```
   ┌──────────────────┐     'TG'        ┌──────────────────────────┐
1: │     n ( ≥ 0 )    │  ──────────>  1:│  FS of TAN(X) of degree n│
   └──────────────────┘                 └──────────────────────────┘
```

or:

```
   ┌──────────────────┐     'TG'        ┌──────────────────┐
1: │    FS of F(X)    │  ──────────>  1:│ FS of TAN(F(X))  │
   └──────────────────┘                 └──────────────────┘
```

In the latter case, the F.S. of `TAN(F(X)` is obtained with the same degree as that of F(X).

<u>N.B</u>: 'TG' calls programs 'CS' 'SN' and 'QFS'.

'TG':(Checksum: # 54897d, Size: 39 bytes)

```
«   DUP   SN   SWAP   CS   QFS   »
```

<u>Examples</u>:

* in 8 to 9 seconds:

```
   ┌─────────┐    'TG'      ┌──────────────────────────────────────────┐
1: │    5    │  ───────> 1: │[0 1 0 .333333333333 0 .133333333334]     │
   └─────────┘              └──────────────────────────────────────────┘
```

As `TAN(X) = X + (1/3)*X^3 +(2/15)*X^5 + o(X^5).`

* in 14 seconds:

```
   ┌─────────────┐   'TG'      ┌──────────────────────────────┐
1: │ [ 0 3 2 -1] │ ──────> 1:  │ [   0   3   2   7.99999999999   ] │
   └─────────────┘             └──────────────────────────────┘
```

As `TAN( 3*X +2*X^2 -X^3 +o(X^3) ) = 3*X +2*X^2 +8*X^3 +o(X^3) .`

FINITE SERIES OF SINH(X) AND HYPERBOLIC SINE OF AN F.S.

'SH' lets you calculate the finite series of `SINH(X)` of degree n (n ≥ 0), or the finite series of `SINH(F(X)` where F is entirely expressed by its finite series.

The functional diagram is as follows:

```
1: | n ( ≥ 0 ) |  --'SH'-->  1: | FS of SINH(X) of degree n |
```

or:

```
1: | FS of F(X) |  --'SH'-->  1: | FS of SINH(F(X)) |
```

In the latter case, the F.S. of `SINH(F(X)` is obtained with the same degree as that of F(X).

N.B: 'SH' calls programs 'DIM' 'CPFS' and 'EX'.

'SH':(Checksum: # 49613d, Size: 104 bytes)

```
« « DUP EX SWAP NEG EX - 2 /
  »
  IF OVER TYPE THEN
     OVER DIM / FLOOR SWAP EVAL CPFS
  ELSE EVAL
  END
»
```

Examples:

* in one second:

```
1: | 4 |  --'SH'-->  1: | [ 0 1 0 .166666666667 0 ] |
```

* in 5 to 6 seconds:

```
1: | [0 3 1 -1 2] |  --'SH'-->  1: | [0 3 1 3.50000000001 6.50000000001] |
```

As `( 3*X +X^2 -X^3 +2*X^4 +o(X^4) )`
`= 3*X +X^2 +3.5*X^3 +6.5*X^4 +o(X^4).`

FINITE SERIES OF COSH(X) AND HYPERBOLIC COSINE OF AN F.S.

'CH' lets you calculate the finite series of `COSH(X)` of degree n `(n ≥ 0)`, or the finite series of `COSH(F(X)` where F is entirely expressed by its finite series.

The functional diagram is as follows:

```
1: [ n ( ≥ 0 ) ]   --'CH'-->  1: [ FS of COSH(X) of degree n ]
```

or:

```
1: [ FS of F(X) ]  --'CH'-->  1: [ FS of COSH(F(X)) ]
```

In the latter case, the F.S. of `COSH(F(X)` is obtained to the maximum possible degree, given the index p of the F.S. of F(X).

N.B: 'CH' calls programs 'DIM' 'CPFS' and 'EX'.

'CH':(Checksum: # 25150d, Size: 109 bytes)

```
«  «   DUP   EX   SWAP   NEG   EX   +   2   /
   »
   IF   OVER   TYPE   THEN
        OVER   DIM   /   FLOOR   1   +   SWAP   EVAL   CPFS
   ELSE   EVAL
   END
»
```

Examples:

* in one second:

```
1: [ 5 ]  --'CH'-->  1: [ [ 1  0 .5  0 4.16666666667E-2  0 ] ]
```

* in 7 to 8 seconds:

```
1: [ [0 0 2 -1 3 1 1 ] ]  --'CH'-->  [ [ 1 0 0 0 2 -2 6.5 -1 6.16666666667 ] ]
```

As `COSH( 2*X^2 -X^3 +3*X^4 +X^5 +X^6 +o(X^6) )`
`= 1 +2*X^4 -2*X^5 +6.5*X^6 -X^7 +6.16666666667*X^8 + o(X^8).`

FINITE SERIES OF TANH(X) AND HYPERBOLIC TANGENT OF AN F.S.

'TH' lets you calculate the finite series of `TANH(X)` of degree n (n ≥ 0), or the finite series of `TANH(F(X)` where F is entirely expressed by its finite series.

The functional diagram is as follows:

```
    ┌───────────────────┐     'TH'        ┌──────────────────────────────┐
1:  │    n ( ≥ 0 )      │  ─────────>  1: │  FS of TANH(X) of degree n   │
    └───────────────────┘                 └──────────────────────────────┘
```

or:

```
    ┌───────────────────┐     'TH'        ┌────────────────────┐
1:  │    FS of F(X)     │  ─────────>  1: │  FS of TANH(F(X))  │
    └───────────────────┘                 └────────────────────┘
```

In the latter case, the F.S. of `TANH(F(X)` is obtained with the same degree as that of F(X).

N.B: 'TH' calls programs 'CH' 'SH' and 'QFS'.

'TH':(Checksum: # 5325d, Size: 39 bytes)

```
« DUP SH SWAP CH QFS »
```

Examples:

* in 6 seconds:

```
    ┌─────────┐    'TH'        ┌──────────────────────────────────────────────┐
1:  │    5    │  ──────>   1:  │[0 1 0 -.333333333333 0 .133333333334]        │
    └─────────┘                └──────────────────────────────────────────────┘
```

As `TANH(X) = X - (1/3)*X^3 +(2/15)*X^5 + o(X^5).`

* in 16 seconds:

```
    ┌──────────────┐    'TH'        ┌────────────────────────────────────┐
1:  │[0 2 -1 3 1]  │  ──────>   1:  │ [  0  2 -1  .333333333334  5 ]     │
    └──────────────┘                └────────────────────────────────────┘
```

As `TANH( 2*X -X^2 +3*X^3 +X^4 +o(X^3) )`
`= 2*X -X^2 +(1/3)*X^3 +5*X^4 +o(X^4) .`

FINITE SERIES OF ARCSIN(X) AND ARC SINE OF AN F.S.

'ASNS' lets you calculate the finite series of `ARCSIN(X)` of degree n (n ≥ 0), or the finite series of `ARCSIN(F(X)` where F is entirely expressed by its finite series.

The functional diagram is as follows:

```
1: [ n ( ≥ 0 ) ]  --'ASNS'-->  1: [ FS of ARCSIN(X) of degree n ]
```

or:

```
1: [ FS of F(X) ]  --'ASNS'-->  1: [ FS of ASIN(F(X)) ]
```

In the latter case, the F.S. of `ARCSINH(F(X)` is obtained with the same degree as that of F(X). The coefficient F(0) must be zero.

N.B: 'ASNS' calls programs 'DIM' and 'CPFS'.

'ASNS': (Checksum: # 63273d, Size: 221 bytes)

```
«  «   →   n
       «   0    n    FOR   i
                0    i    DUP   2   /   COMB   2   i   ^   /   i   1   +   /
           2    STEP
           n    2    MOD    NOT    DROPN
           n    1    +    →ARRY
       »
   »
   IF    OVER    TYPE    THEN
         OVER    1    GET
         IF    ABS    THEN    DROP    "Error"
         ELSE    OVER    DIM    /    FLOOR    SWAP    EVAL    CPFS    END
   ELSE    EVAL
   END
»
```

Examples:

* in one second:

```
1: [ 6 ]  --'ASNS'-->  1: [0  1  0  .166666666667  0  .075  0]
```

* in 5 seconds:

```
1: [0 1 -3 2 5]  --'ASNS'-->  1: [ 0  1  -3  2.16666666667  3.5 ]
```

As `ASIN( X -3*X^2 +2*X^3 +5*X^4 +o(X^4) )`
`= X -3*X^2 +2.16666666667*X^3 +3.5*X^4 +o(X^4).`

FINITE SERIES OF ARCCOS(X) AND ARC COSINE OF AN F.S.

'ACS' lets you calculate the finite series of ARCCOS(X) of degree n (n ≥ 0), or the finite series of `ARCCOS(F(X)` where F is entirely expressed by its finite series.

The functional diagram is as follows:

```
1: |  n ( ≥ 0 ) |  |   'ACS'  ---->  1: | FS of ARCCOS(X) of degree n |
```

or:

```
1: |  FS of F(X)  |   'ACS'  ---->  1: | FS of ACOS(F(X)) |
```

In the latter case, the F.S. of `ARCCOS(F(X)` is obtained with the same degree as that of F(X). The coefficient F(0) must be zero.

N.B: 'ACS' calls program 'ASNS'.

'ACS':(Checksum: # 63964d, Size: 42.5 bytes)

```
«  ASNS  NEG  1  π  →NUM  2  /  PUT  »
```

Examples:

* in one second:

```
1: |  5  |  'ACS' ---->  | [1.5707963268 -1 0 -.166666666667 0 -.075 ] |
```

As `ARCCOS(X)= π/2 -X -(1/6)*X^3 -(3/40)*X^5 +o(X^5).`

* in 5 seconds:

```
1: | [0 1 -1 2 1] |  'ACS' ---->  1: |   [  1.5707963268     -1     1
                                         -2.16666666667   -.499999999999 ] |
```

As
```
ARCCOS( X -X^2 +2*X^3 +X^4 +o(X^4) )
  = π/2 -X +X^2 -(13/6)*X^3 -(1/2)*X^4 +o(X^4).
```

FINITE SERIES OF ARCTAN(X) AND ARC TANGENT OF AN F.S.

'ATG' lets you calculate the finite series of `ARCTAN(X)` of degree n `(n ≥ 0)`, or the finite series of `ARCTAN(F(X)` where F is entirely expressed by its finite series.

The functional diagram is as follows:

```
1: [ n ( ≥ 0 ) ]   --'ATG'-->  1: [ FS of ATAN(X) of degree n ]
```

or:

```
1: [ FS of F(X) ]  --'ATG'-->   : [ FS of ATAN(F(X)) ]
```

In the latter case, the F.S. of `ARCTAN(F(X)` is obtained with the same degree as that of F(X). The coefficient F(0) must be zero.

N.B: 'ATG' calls programs 'DIM' and 'CPFS'.

'ATG': (Checksum: # 55351d, Size: 213 bytes)

```
«  «    →  n
        «  0   n   FOR  i
                   0   i   1   +   INV
                   0   i   3   +   INV   NEG
           4   STEP
           3   n   4   MOD   -   DROPN
           n   1   +   →ARRY
        »
   »
   IF    OVER   TYPE   THEN
         OVER   1   GET
         IF   ABS   THEN   DROP   "Error"
         ELSE   OVER   DIM   /   FLOOR   SWAP   EVAL   CPFS   END
   ELSE   EVAL
   END
»
```

Examples:

* in one second:

```
1: [ 6 ]   --'ATG'-->  1: [ 0  1  0  -.333333333333  0  .2  0 ]
```

* in 5 seconds:

```
1: [0 3 1 -1 2]   --'ATG'-->  1: [0 3 1 -9.99999999999 -6.99999999999]
```

As `ATAN( 3*X +X^2 -X^3 +2*X^4 +o(X^4) )`
`= 3*X +X^2 -10*X^3 -7*X^4 +o(X^4).`

FINITE SERIES OF ARCSINH(X) AND HYPERBOLIC ARC SINE OF AN F.S.

'ASH' lets you calculate the finite series of `ARCSINH(X)` of degree n (n ≥ 0), or the finite series of `ARCSINH(F(X)` where F is entirely expressed by its finite series.

The functional diagram is as follows:

```
1: [ n ( ≥ 0 ) ]   --'ASH'-->   1: [FS of ARCSINH(X) of degree n]
```

or:

```
1: [ FS of F(X) ]   --'ASH'-->   1: [ FS of ASINH(F(X)) ]
```

In the latter case, the F.S. of `ARCSINH(F(X)` is obtained with the same degree as that of F(X). The coefficient F(0) must be zero.

N.B: 'ASH' calls programs 'ASNS' and 'X→AX'.

'ASH': (Checksum: # 25447d, Size: 128 bytes)

```
«   IF   DUP   TYPE   THEN
                (0,1)   *   ASNS
          ELSE
                ASNS   (1,0)   *   (0,1)   X→AX
    END
    IM
»
```

Examples:

* in 2 seconds:

```
1: [ 6 ]   --'ASH'-->   1: [0  1  0  -.166666666667  0  .075  0]
```

* in 6 to 7 seconds:

```
1: [0 1 -3 2 5]   --'ASH'-->   1: [  0  1  -3  1.83333333333  6.5  ]
```

As `ASINH( X -3*X^2 +2*X^3 +5*X^4 +o(X^4) )`
`= X -3*X^2 +(11/6)*X^3 +(13/2)*X^4 +o(X^4).`

FINITE SERIES OF ARCTANH(X) AND HYPERBOLIC ARC TANGENT OF AN F.S.

'ATH' lets you calculate the finite series of `ARCTANH(X)` of degree n ($n \geq 0$), or the finite series of `ARCTANH(F(X)` where F is entirely expressed by its finite series.

The functional diagram is as follows:

```
   ┌──────────────────┐     'ATH'          ┌──────────────────────────────┐
1: │     n ( ≥ 0 )    │   ─────────>    1: │FS of ARCTANH(X) of degree n  │
   └──────────────────┘                    └──────────────────────────────┘
```

or:

```
   ┌──────────────────┐     'ATH'           ┌───────────────────┐
1: │    FS of F(X)    │   ─────────>     1: │ FS of ATANH(F(X)) │
   └──────────────────┘                     └───────────────────┘
```

In the latter case, the F.S. of `ARCTANH(F(X)` is obtained with the same degree as that of F(X). <u>The coefficient F(0) must be zero.</u>

<u>N.B</u>: 'ATH' calls programs 'ATG' and 'X→AX'.

'ATH':(Checksum: # 59212d, Size: 126 bytes)

```
«   IF   DUP   TYPE   THEN
               (0,1)   *   ATG
         ELSE
               ATG    (1,0)    *    (0,1)    X→AX
    END
    IM
»
```

<u>Examples</u>:

* in 2 seconds:

```
   ┌──────────┐    'ATH'     ┌──────────────────────────────────────────┐
1: │    6     │  ────────> 1:│ [ 0  1  0  .333333333333  0  .2  0 ]     │
   └──────────┘              └──────────────────────────────────────────┘
```

* in 6 seconds:

```
   ┌─────────────┐   'ATH'      ┌──────────────────────────────────────┐
1: │[0 3 1 -1 2] │ ────────> 1: │  [  0  3  1  7.99999999999  11  ]    │
   └─────────────┘              └──────────────────────────────────────┘
```

As `ATANH( 3*X +X^2 -X^3 +2*X^4 +o(X^4) )`
`= 3*X +X^2 +8*X^3 +11*X^4 +o(X^4).`

GEOMETRY

The 'GEOM' directory contains programs for affine or Euclidean geometry.

These programs are used mainly to find the equation of a set of points (straight line, plane, circle, sphere) whose main characteristics are already known, or to calculate distances or angular distances.

Here is the list of programs in the 'GEOM' directory:

'STRT' : equation of a straight line for which:
* two points
or * one point and a direction vector are known.

'PLAN' : equation of a plane for which:
* three points
or * two points and a direction vector
or * one point and two direction vectors are known.

'CIRC' : equation of a circle or sphere for which:
* the centre and the radius
or * two diametrically opposite points are known.

'DIST' : calculates the distance between:
* a point and a straight line (in 2 or 3 dimensions)
or * a point and a plane
or * two straight lines (in space).

'ANGL' : calculates the angular distance between:
* two vectors
or * two straight lines
or * two planes.

EQUATION OF A STRAIGHT LINE

'STRT' lets you find, in symbolic form, the equation of a straight line S in the plane for which:

- * two separate points P and Q
- or * a point P and a direction vector u are known.

P and Q (or P and u) must be entered at levels 1 and 2.

A point with coordinates x and y is represented by the vector `[x,y,1]`. A vector whose components are `x` and `y` is represented by `[ x, y, 0 ]`.
It is therefore the third component that determines whether we are dealing with a point or a vector.

The equation is obtained in symbolic form: `'A*X+B*Y+C=0'`.

'STRT': (Checksum: # 63312d, Size: 65 bytes)

```
«   CROSS   V→   ROT   'X'   *   ROT
    'Y'   *   +   SWAP   +   0   =
»
```

Example 1:

(in one second)
The equation of the straight line passing through the points `P(-1,3)` and `Q(2,5)`.

```
2: [ -1  3  1 ]     'STRT'     2:
                  -------->
1: [  2  5  1 ]                1: '-(2*X)+3*Y-11=0'
```

Example 2:

(in one second)
Equation of the straight line passing through the point `P(3,2)` and the direction vector `u(4,1)`.

```
2: [  3  2  1 ]     'STRT'     2:
                  -------->
1: [  4  1  0 ]                1: '-X+4*Y-5=0'
```

In the example above, the order in which P and u are given may be reversed.

EQUATION OF A PLANE

'PLAN' lets you find, in symbolic form, the equation of a plane (π) for which:

- * three points P, Q and R
- or * two points P and Q and a direction vector u
- or * a point P and two direction vectors u and v are known.

A point whose coordinates are x, y and z is represented by the vector `[ x y z 1 ]`.
A vector whose components are x, y and z is represented by the vector `[ x y z 0 ]`.
It is therefore the fourth component that determines whether we are dealing with a vector or a point.

The three elements P, Q and R (or P, Q and u or P, u and v) must be entered at levels 1, 2 and 3 of the stack, in any order.

The equation is obtained at level 1 in symbolic form, i.e.: `'A*X+B*Y+C*Z+D=0'`.

'PLAN': (Checksum: # 63868d, Size: 166.5 bytes)

```
«  { X  Y  Z  1 }   →   a   b   c   L
   «   0   0   0   0   a   V→   b   V→   c   V→
       { 4  4 }   →ARRY   0
       1   4   FOR   k
           OVER   k   1   PUT   DET   L   k   GET   *   +
       NEXT
       0   =   SWAP   DROP
   »
»
```

Example 1:

Equation of the plane passing through the points:
`P(-1,2,3) Q(4,1,5)` and `R(2,-3,0)`:
(in 3 seconds)

	Before			After
3:	`[ -1 2 3 1 ]`		3:	
2:	`[ 4 1 5 1 ]`	`'PLAN'` →	2:	
1:	`[ 2 -3 0 1 ]`		1:	`'13*X+21*Y-22*Z+37=0'`

Example 2:

Equation of the plane passing through the point P(-1,1,-3) and with direction vectors u(-2,1,3) and v(-1,4-2).
(in 3 seconds)

	Before			After
3:	`[ -2 1 3 0 ]`		3:	
2:	`[ -1 1 -3 1 ]`	`'PLAN'` →	2:	
1:	`[ -1 4 -2 0 ]`		1:	`'14*X+7*Y+7*Z+28=0'`

EQUATION OF A CIRCLE OR A SPHERE

'CIRC' lets you find, in symbolic form, the equation of a circle (C) in two-dimensional space, or of a sphere (S) in three-dimensional space for which:
* the centre Ω and the radius R

or * two diametrically opposite points are known.

Here, a point is represented by the vector of its coordinates `[x,y]` (on a surface) or `[x y z]` (in space).
The radius R is a positive real number.

The two elements Ω and R (or A and B) must be entered at levels 1 and 2, in any order. The equation is therefore obtained at level 1 in symbolic form, i.e.:

`'X^2+Y^2+Z*X+B*Y+C=0'` for a circle.
`'X^2+Y^2+Z^2+A*X+B*Y+C*Z+D=0'` for a sphere.

'CIRC': (Checksum: # 31394d, Size: 311.5 bytes)

```
«   DUP2   TYPE   SWAP   TYPE   IF   <   THEN   SWAP   END
    DUP   SIZE   1   GET   →   b   a   d
    «   IF   b   TYPE   3   ==   THEN
            b   a   +   2   /   b   a   -   2   /
            ABS   'b'   STO   'a'   STO
        END
        0   1   d   FOR   k
            "'"   87   k   +   CHR   +   OBJ→   2   ^   +
        NEXT
        1   d   FOR   k
            a   k   GET   2   *   NEG   "'"
            87   k   +   CHR   +   OBJ→   *   +
        NEXT
        a   DUP   DOT   b   SQ   -   +   0   =
    »
»
```

<u>Example 1</u>: (in two seconds)

Equation of the circle with centre `Ω(2,3)` and a radius of 5.

```
2:  [ 2 3 ]      'CIRC'      2:
1:     5       ---------->   1:  'X^2+Y^2-4*X-6*Y-12=0'
```

<u>Example 2</u>: (in 3 to 4 seconds)

Equation of the sphere with diameter AB where `A = (1,1,1)` and `B = (3,5,5)`.

```
2:  [ 1  1  1 ]   'CIRC'   2:
1:  [ 3  5  5 ]  ------->  1:  'X^2+Y^2+Z^2-4*X-6*Y-6*Z+13=0'
```

CALCULATING DISTANCES

====================

'DIST' calculates the distance between:

* a point A(x,y) and a straight line given by its equation (in two-dimensional space)

or * a point A(x,y,z) and a plane given by its equation

or * a point and a straight line given by a point and a direction vector (in two or three-dimensional space)

or * two straight lines each given by a point and a direction vector (in space).

A point `A(x,y)` is represented by the vector `[ x y ]`.
A point `A(x,y,z)` is represented by the vector `[ x y z ]`.
The straight line of the equation `'ax+by+c=0'` is represented by `[ a b c ]`.
The plane `'ax+by+cz+d=0'` is represented by `[ a b c d ]`.

When a straight line is given by a point `A(x,y,z)` (or, in two dimensions, `A(x,y)`) and a direction vector `u(a,b,c)` (or `u(a,b)`), it is represented in the stack by a list in the following order:

`{ [ x y z ] [ a b c ] }`
(or `{ [ x y ] [ a b ] }`).

The two objects between which we want to calculate the distance are entered at levels 1 and 2 of the stack. If one of them is a point, it must be entered a level 2.
The distance calculated is given at level 1.

'DIST': (Checksum: # 49003d, Size: 344.5 bytes)

```
«   DUP2    TYPE    SWAP    TYPE
    →   t1   t2
    «   CASE
            t1   3   ==   t2   3   ==   AND   THEN
                 →   p   e
            «    e   p   SIZE   RDM   DUP   p   DOT   e
                 DUP   SIZE   GET   +   ABS   SWAP   ABS   /
            »
            END
            t2   3   ==   t1   5   ==   AND   THEN
                 →   p   d
                 «   d   EVAL   DUP   ROT   p   -
                     CROSS   ABS   SWAP   ABS   /
                 »
            END
            →   d1   d2
            «   d1   2   GET   d2   2   GET   DUP2   V→
                4   ROLL   V→   d1   1   GET
                d2   1   GET   -   V→   { 3 3 }   →ARRY
                DET   ABS   3   ROLLD   CROSS   ABS   /
            »
        END
    »
»
```

PROGRAM 'DIST': PRACTICAL EXAMPLES

Example 1: (in under one second)

Distance from the point M(1,-2) to the straight line 3x+4y+1=0.

```
2: [ 1 -2 ]                'DIST'   2:
1: [ 3  4  1  ]          ------->   1:        8
```

Example 2: (in one second)

Distance from the point M(2,-1,3) to the plane x+2y+2z-5=0.

```
2: [ 1 -2 ]                'DIST'   2:
1: [ 3  4  1  ]          ------->   1:        8
```

Example 3: (in under one second)

Distance from the point M(1,-2) to the straight line given by the point A(5,-4) and the direction vector u(-4,3). The data is identical to example 1.

```
2: [  2 -1  3 ]            'DIST'   2:
1: [ 1  2  2 -5 ]        ------->   1: .333333333333
```

Example 4: (in under one second)

Distance from the point M(3,-2,1) to the straight line S given by the point A(-1,1,4) and the direction vector u(2,-1,5).

```
2: [ 1 -2 ]                     'DIST'   2:
1: { [5 -4] [-4 3] }          ------->   1:      0.8
```

Example 5: (in one second)

Distance from the straight line S given by the point A(1,0,4) and the vector u(2,3,5) and the straight line D' given by the point B(-1,-5,2) and the vector v(4,3,0).

```
2: [ 3  -2  1 ]                          'DIST'   2:
1: { [-1 1 4 ] [ 2 -1 5 ] }            ------->   1: 5.78503817331
```

CALCULATING ANGULAR DISTANCE

'ANGL' calculates the angular distance (θ ($0 \le \theta \le \pi/2$) between:

* two vectors u and v (in two or three-dimensional space)

or * two straight lines with known direction vectors u and v (in two or three-dimensional space)

or * two known planes given by their equations.

A vector u(a,b) (or, in three dimensions, u(a,b,c)) is represented by [a b] (or [a b c]).
[a b] (or [a b c]) also enables us to designate any straight line $ax+by+c=0$ (or any plane $ax+by+cz+d=0$).

The two objects between which we want to calculate the angular distance must be entered at levels 1 and 2.

The angular distance θ is then obtained in radians, degrees (in HMS format) and gradians. The 3 results are given to four decimal places in the form of signed real numbers (see the example below).

'ANGL': (Checksum: # 26751d, Size: 122 bytes)

```
«   RAD    4      FIX    DUP2    DOT    ABS    SWAP
    ABS    ROT    ABS    *       /      ACOS   "radians"   →TAG
    DUP    R→D    →HMS   "deg( HMS )"   →TAG
    DUP    HMS→   .9     /       "gradians"    →TAG
»
```

Example:

Angular distance between the vectors u(1,-4,2) and v(3,1,5) (in one second):

2:	[1 -4 2]	'ANGL'	4:	
		——————>	3:	radians: 1.2324
			2:	deg(HMS): 70.3642
1:	[3 1 5]		1:	gradians: 78.4573

DIFFERENTIAL GEOMETRY

The 'DIFG' directory contains programs on differential geometry. This involves resolving geometrical problems where derivatives and primitives have to be calculated.

This is an area in which the HP48 can be used to program relatively complex problems with ease (problems that would prove tricky to resolve with a popular programming language like Turbo Pascal).

What these programs do is exploit the HP48's ability to integrate and above all to differentiate a function written in symbolic form. It is particularly useful to be able to obtain expressions of the partial derivatives of a function (with respect to any of its variables).

Here is the list of programs in the 'DIFG' directory:

'RECTP' : Rectifying (finding the length of) a plane curve (given by the equation `Y=F(X)`, or in polar coordinates or by the parametric equations `X=X(T)`, `Y=Y(T)`).

'RECTS' : Rectifying (finding the length of) a space curve (three-dimensional curve) given by parametric equations.

'LINT' : Calculates a line integral in two or three-dimensional space along an arc.

'AREA' : Area of a two-dimensional domain limited by a closed curve that is given by parametric equations (with polar or Cartesian coordinates).

'CVTRE' : Calculates the radius of curvature and the centre of curvature of a point on a plane curve.

'DIVRG' : Calculates the divergence of a vector field.

'CURL' : Calculates the curl of a vector field.

'GRADI' : Gradient of a function of several variables.

'LAPL' : Laplacean of a function of several variables.

'DIFF' : Calculates the differential of a function of several variables.

RECTIFYING A PLANE CURVE

'RECTP' calculates the approximate length of a plane curve defined in either of three ways:

* by a Cartesian equation `Y=F(X)`, where `A ≤ X ≤ B`.
* by parametric equations `X=X(T)`, `Y=Y(T)`, where `A≤T≤B`.
* by a polar equation `RO=RO(T)`, where `A ≤ T ≤ B`.

The functional diagram is as follows:

```
               'RECTP'
1: [ list ]  ----------->  1: [ length ]
```

where "list" is a list of the elements required for computation and "length" is an approximate value of the result.

"List" is ordered as follows:

`{ F(X) A B }`	in the first case,
`{ X(T) Y(T) A B }`	in the second case,
`{ RO(T) A B }`	in the last case,

where:

`A` is the lower value of the parameter (`X` or `T`).
`B` is the upper value of the parameter (`X` or `T`).
`F(X)` is the expression, in symbolic form, of the variable '`X`'.
`X(T)`, `Y(T)` and `RO(T)` are expressions, in symbolic form, of the variable '`T`'.

Integrals are calculated to the degree of accuracy specified by the display mode:

* In **n FIX** mode, integrals calculated are correct to n decimal places.
* In **STD** mode, integrals are calculated as accurately as possible (which may take time).

The value of the variable **IERR** obtained is greater than the absolute error made.

'RECTP': (Checksum: # 62297d, Size: 269.5 bytes)

```
«  OBJ→    { X  T }    PURGE
   →   a    b    n
   «   IF   n    4    ==     THEN
                 'T'   ∂    2    ^    SWAP   'T'   ∂    2   ^   +   √   'T'
            ELSE
                 DUP    'T'    ∂
                 IF    DUP    0    SAME    THEN
                            DROP    'X'    ∂    2    ^    1    +    √    'X'
                       ELSE
                            2    ^    SWAP    2    ^    +    √    'T'
                 END
       END
       a    b    4    ROLL    4    ROLL    ∫    →NUM
   »
»
```

PROGRAM 'RECTP': PRACTICAL EXAMPLES

Example 1:

Length of the catenary `Y=CH(X)`, where `0≤X≤1`.
(Having first switched to 5 FIX mode)

```
2:                             'RECTP'   2:
                               ------>
1:   { 'COSH(X)' 0 1 }                   1:        1.17520
```

The result is calculated within approximately 6 seconds. The exact result shown is:

`SH(1)= ( e - 1/e )/2 ≈ 1.17520119364.`

Example 2:

Length of the cycloid `X=COS(T), Y=T-SIN(T), 0≤T≤2π.`
(Having first switched to 3 FIX mode)

```
2:                                                    'RECTP'
                                                      ------>
1:   { 'COS(T)' 'T-SIN(T)' 0  '2*π' }                      1:   8.000
```

The result is calculated within approximately 9 seconds. The exact result is 8.

Example 3:

Length of the cardioid `RO=1+COS(T)`.
(Having first switched to 4 FIX mode)
To obtain the total length, T must vary between 0 and 2π.

```
2:                                    'RECTP'
                                      ------>
1:   {  '1+COS(T)' 0 '2*π'  }              1:        8.0000
```

The result is calculated within approximately 9 seconds. The exact result is 8.

RECTIFYING A SPACE CURVE

'RECTS' calculates the approximate length of a space curve defined in either of two ways:

* by the parametric equations `X=X(T), Y=Y(T), Z=Z(T),` where T takes all the values on the segment [A, B].
* by the parametric equations with cylindrical coordinates `RO=RO(T), Z=Z(T),` where `A ≤ T ≤ B,.`

The functional diagram is as follows:

```
     ┌──────────┐      'RECTS'          ┌──────────┐
  1: │   list   │   ──────────────>  1: │  length  │
     └──────────┘                       └──────────┘
```

where "list" is a list of the elements required for computation and "length" is an approximate value of the result.

"List" is ordered as follows:
`{ X(T) Y(T) Z(T) A B }` in the first case,
`{ RO(T) Z(T) A B }` in the second case,

where:
`A` is the lower bound of the parameter `T`.
`B` is the upper bound of the parameter `T`.
`X(T), Y(T), Z(T)` and `RO(T)` are expressions, in symbolic form, of the variable `'T'`.

Integrals are calculated to the degree of accuracy specified by the display mode:

* In **n FIX** mode, integrals calculated are correct to n decimal places.
* In **STD** mode, integrals are calculated as accurately as possible (which may take time).

The value of the variable **IERR** is greater than the absolute error made.

'RECTS': (Checksum: # 41324d, Size: 203.5 bytes)

```
«  OBJ→    'T'    PURGE
   →   a    b    n
   «   'T'    ∂    2    ^    SWAP
       IF    n    5    ==    THEN
                 'T'    ∂    2    ^    +    SWAP    'T'    ∂    2    ^
             ELSE
                 DUP    'T'    ∂    2    ^    SWAP    2    ^    +
       END
       +    √    a    b    ROT    'T'    ∫    →NUM
   »
»
```

PROGRAM 'RECTS': PRACTICAL EXAMPLES

Example 1:

Length of the arc given by the parametric equations:
`Y=X², Z=(2/3)X^3`, where $0 \le X \le 1$.
Here, we take X as the parameter T.
We first switch to 4 FIX mode.

2:		'RECTS'		
1:	{ 'T' 'T^2' '2*T^3/3' 0 1 }	——>	1:	1.6667

The result is obtained within 10 seconds.
The exact result is $5/3 \approx 1.66666666667$.

Example 2:

Length of the circular helix given in cylindrical coordinates by: `RO=COS(T), Z=T`, where
$0 \le T \le 2\pi$.
We first switch to 4 FIX mode.

2:		'RECTS'		
1:	{ 'COS(T)' 'T' 0 '2*π' }	——>	1:	8.8858

The result is obtained within 4 seconds.
The exact result is $2\sqrt{2\pi} \approx 8.8857658763$.

CALCULATING LINE INTEGRALS

'LINT' calculates the line integral:

1) of the differential form
   `w = P(X,Y)dX + Q(X,Y)dY`
   along the arc `X=X(T), Y=Y(T)`, where `A ≤ T ≤ B`.
2) of the differential form
   `w = P(X,Y,Z)dX + Q(X,Y,Z)dY + R(X,Y,Z)dZ`
   along the arc `X=X(T), Y=Y(T), Z=Z(T)`, where `A ≤ T ≤ B`.

   The functional diagram is as follows:

```
1: [ list ]  --'LINT'-->  1: [ integral ]
```

where "list" is a list containing the elements required for computation and "integral" is an approximate value of the result.

"List" is ordered as follows:

`{P(X,Y) Q(X,Y) X(T) Y(T) Z(T) A B }` in the first case,
`{P(X,Y,Z) Q(X,Y,Z) R(X,Y,Z) X(T) Y(T) Z(T) A B }` in the second case,
where:

`A` is the lower bound of the parameter `T`.
`B` is the upper bound of the parameter `T`.

`P(X,Y)` and `Q(X,Y)` are expressions, in symbolic form, in terms of `X` and `Y`.
`P(X,Y,Z), Q(X,Y,Z)` and `R(X,Y,Z)` are expressions, in symbolic form, in terms of `X, Y` and `Z`.
`X(T), Y(T)` and `Z(T)` are expressions, in symbolic form, of the variable '`T`'.

Integrals are calculated to the degree of accuracy specified by the display mode:

* In **n FIX** mode, integrals calculated are correct to n decimal places.
* In **STD** mode, integrals are calculated as accurately as possible (which may take time).

The value of the variable **IERR** obtained is greater than the absolute error made.

'LINT': (Checksum: # 58416d, Size: 282.5 bytes)

```
«  OBJ→    'T'    PURGE
   →   a    b    n
   «   IF    n    8    ==    THEN    'Z'    STO    END
        'Y'    STO    'X'    STO
       IF    n    8    ==    THEN
                 'Z'    'T'    ∂    *
            ELSE    0
       END
       3    ROLLD    'Y'    'T'    ∂    *    SWAP
       'X'    'T'    ∂    *    +    +
       a    b    ROT    'T'    SHOW    'T'    ∫    →NUM
       { X   Y   Z }    PURGE
»  »
```

PROGRAM 'LINT': PRACTICAL EXAMPLES

Example 1:

We want to calculate the line integral:

$$\int_{\Gamma} (2-y)dx + xdy \ ,$$

where Γ is the cycloid:
`X=T-SIN(T), Y=1-COS(T), 0 ≤ T ≤ 2*π.`

We first switch to 4 FIX mode.

We then enter the list:

```
{ '2-Y' 'X' 'T-SIN(T)' '1-COS(T)' 0 '2*π' }
```

at level 1 of the stack and call 'LINT'.

Within 17 seconds, we obtain:

```
1: -6.2832
```

The exact result being:

$-2\pi \approx -6.28318530718.$

Example 2:

We want to calculate the line integral:

$$\int_{\Gamma} (y-z)dx + (z-x)dy + (x-y)dz,$$

where Γ is the spiral helix `X=COS(T), Y=SIN(T), Z=T, 0≤T≤2π.`

We first switch to 4 FIX mode.
We then enter the list:

```
{ 'Y-Z' 'Z-X' 'X-Y' 'COS(T)' 'SIN(T)' 'T' 0 '2*π' }
```

at level 1 of the stack and call 'LINT'.

Within 19 seconds, we obtain:

```
1: -12.5664
```

The exact result being:

$-4\pi \approx -12.5663706144.$

CALCULATING THE AREA UNDER A PLANE CURVE

'AREA' calculates the area under a plane closed curve Γ given by:

1) the parametric equations `X=X(T), Y=Y(T), A ≤ T ≤ B.`
2) polar coordinates `RO=RO(T)`, where `A ≤ T ≤ B.`

The functional diagram is as follows:

```
     ┌─────────────┐      'AREA'          ┌─────────────┐
  1: │    liste    │   ───────────>   1:  │    area     │
     └─────────────┘                      └─────────────┘
```

where "list" is a list containing the elements required for computation and "area" is an approximate value of the result.

"List" is ordered as follows:

`{ X(T) Y(T) A B }` in the first case,
`{ RO(T) A B }` in the second case,

where:

A is the lower bound of the parameter T.
B is the upper bound of the parameter T.
X(T), Y(T) and RO(T) are expressions, in symbolic form, of the variable 'T'.

Integrals are calculated to the degree of accuracy specified by the display mode:

* In **n FIX** mode, integrals calculated are correct to n decimal places.
* In **STD** mode, integrals are calculated as accurately as possible (which may take time).

The value of the variable **IERR** obtained is greater than the absolute error made.

Notes:

In the first case, the integral calculated is:

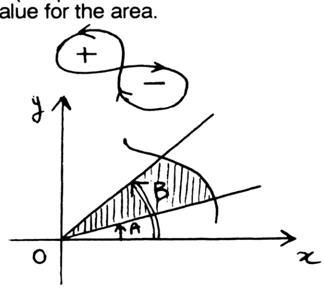
$$\int_{\Gamma} X dY,$$ where Γ is the boundary curve from T=A to T=B.

In the second case, the integral calculated is:

$$(1/2)\int_{A}^{B} RO^2(T)\ dT.$$

The curve Γ from A to B must be in the anti-clockwise direction (the points in the area inside the curve must be to the left), otherwise we obtain a negative value for the area.

If the curve has double points on it and the area inside it has several components, the components covered in the anti-clockwise direction are counted as positive and the others as negative.



The program does not check whether the curve Γ is in fact closed or not. For a curve defined by polar coordinates, and if it is not closed, we obtain the area under the curve Γ and between the half lines with a polar angle A and B.

TEXT OF PROGRAM 'AREA' AND PRACTICAL EXAMPLES

'AREA': (Checksum: # 57317d, Size: 134 bytes)

```
«  OBJ→    'T'    PURGE
   →    a    b    n
   «   IF    n    3    ==    THEN
                  2    ^    2    /
             ELSE
                  'T'    ∂    *
       END
       a    b    ROT    'T'    ∫    →NUM
   »
»
```

Example 1:

We want to find the area inside the ellipse:

$$X^2/4 + Y^2/9 = 1$$

The parametric equations of this ellipse are:

X=2COS(T), Y=3SIN(T), where $0 \le T \le 2\pi$.

We first switch to 5 FIX mode.

		'AREA' →	
	{'2*COS(T)' '3*SIN(T)' 0 '2*π' }	1:	18.84956

We obtain a result within 8 seconds, the exact result being:

$$6\pi \approx 18.8495559215.$$

Example 2:

We want to find the area inside the cardioid RO=1+COS(T) that is described over the segment T = [0,2π].
We first switch to 6 FIX mode.

Using program 'AREA', we find:

2:		'AREA' →	
1:	{ '1+COS(T)' 0 '2*π' }	1:	4.712389

We obtain a result within 15 seconds, the exact result being:

$$3\pi/2 \approx 4.71238898038.$$

CENTRE AND RADIUS OF CURVATURE OF A PLANE CURVE

'CVTRE' calculates the centre of curvature and the radius of curvature at a point on a plane curve given by:

1) the equation `Y=F(X)`
2) the parametric equations `X=X(T), Y=Y(T)`
3) polar coordinates `RO=RO(T)`.

You should proceed as follows:

First, enter at level 1 of the stack:
* the algebraic expression characterizing `F(X)` (first case)
* the list `{ X(T) Y(T) }` characterizing the algebraic expressions `X(T)` and `Y(T)` (second case)
* the algebraic expression characterizing `RO(T)` (third case).

Then call program 'CVTRE'.

Program 'CVTRE' is halted after a moment while the partial derivatives are computed.
A personalized menu is then displayed on the display panel:

```
4:
3:
2:
1:
 →PAR |      | CENT | RADS |      | EXIT
```

To move to a specific point on the curve, we simply enter the value of the parameter at level 1, then press the "→PAR" key.

By pressing "CENT", we then obtain the coordinates `(XR,YR)` of the centre of curvature (in the form of a complex number).

By pressing "RADS", we then obtain the radius of curvature R.

These operations can be repeated for any number of points, during which time program 'CVTRE' is halted.

Press "EXIT" to quit the program.

R and `(XR,YR)` are calculated very quickly (as program 'CVTRE' in fact determines `R`, `XR` and `YR`, expressed symbolically. If we use the corresponding keys on the calculator, the expressions will only be evaluated at the point previously indicated).

N.B: the radius of curvature is calculated for a curve that is described for an increasing value of T.

TEXT OF PROGRAM 'CVTRE'

'CVTRE': (Checksum: # 10389d, Size: 609 bytes)

```
«   'T'   PURGE
    IF   DUP   TYPE   5   ==   THEN
            EVAL
        ELSE
            IF   DUP   'T'   ∂   0   SAME   THEN
                'X'   SWAP   'T'   'X'   STO
                    ELSE
                        DUP   'COS(T)'   *   SWAP   'SIN(T)'   *
            END
    END
    →   x   y
    «   x   'T'   ∂   DUP   'T'   ∂   y   'T'   ∂   DUP   'T'   ∂
        →   x1   x2   y1   y2
        «   x1   y2   *   y1   x2   *   -   x1   SQ   y1   SQ   +
            →   d   n
            «   n   d   /   y1   *   NEG   x   +   n   d   /
                x1   *   y   +   n   1.5   ^   d   /
                →   xr   yr   r
                «   {
                        {   "→PAR"   «   'T'   STO   »   }
                        {   }
                        {   "CENT"
                            «   xr   EVAL   yr   EVAL   R→C   »   }
                        {   "RADS"   «   r   EVAL   »   }
                        {   }
                        {   "EXIT"   CONT   }
                    }
                    TMENU   HALT   { X   T }   PURGE   2   MENU
                »
            »
        »
    »
»
```

PROGRAM 'CVTRE': PRACTICAL EXAMPLES

Example 1:

Radius of curvature and curvature of the catenary `Y = CH(X)`.
We first enter `'COSH(X)'` at level 1 and call 'CVTRE'.
The personalized menu is displayed after two seconds.
If we give a value of `0` to the parameter (here `X`), we find:
Centre of curvature: `(0,2)`, Radius of curvature: 1.
If we give a value of 1 to the parameter, we find (in 5 FIX mode):
Centre of curvature: `( -0.81343, 3.08616 ),`
Radius of curvature: `2.38110.`
Press "EXIT" to quit the program.

Note:here, the theoretical values are at the point on the x-axis:

`XR=X-COSH(X)*SINH(X), YR=2*COSH(X), R=COSH(X)²`.

Example 2:

Radius of curvature and curvature of the cycloid:

`X(T)=T-SIN(T), Y(T)=1-COS(T).`

We first enter `{ 'T-SIN(T)' '1-COS(T)' }` at level 1 and call 'CVTRE'.
The personalized menu is displayed after two seconds.
If we give a value of π to the parameter (here T), we find:
Centre of curvature: `(0,2)`, Radius of curvature: 1.
If we give a value of 1 to the parameter, we find (in 5 FIX mode):
Centre of curvature: `( 3.14159, -2.00000 ),`
Radius of curvature: `-4.00000.`

Note:here, the theoretical values are at the point with parameter T:

`XR=T+SIN(T), YR=-1+COS(T), R=-4*SIN(T/2).`

Example 3:

Radius of curvature and curvature of the cardioid defined by the polar equation

`RO=1+COS(T).`

We first enter `'1+COS(T)'` at level 1 and call 'CVTRE'.
The personalized menu is displayed after 9 seconds.

If we give a value of 0 to the parameter, we find:
Centre of curvature: `( 0.66667, 0.00000),`
Radius of curvature: `1.33333`
If we give a value of $\pi/2$ to the parameter, we find (in 5 FIX mode):
Centre of curvature: `( 0.66667, 0.33333 ),`
Radius of curvature: `0.94281.`

Note:the theoretical value of R at the point with parameter T is: `R=4/3*COS(T/2).`

DIVERGENCE OF A VECTOR FIELD

'DIVRG' calculates the divergence div(E) of a vector field E. The functional diagram is as follows:

```
     ┌──────────────┐                 ┌──────────────┐
  2: │      E       │   'DIVRG'    2: │              │
     ├──────────────┤  ─────────>     ├──────────────┤
  1: │  variables   │              1: │    div(E)    │
     └──────────────┘                 └──────────────┘
```

The field E must be represented here by the list of its components, each of which is an algebraic expression.

"variables" denotes the list of variables with respect to which the partial derivatives are to be calculated.

If we take the example of a field in three-dimensional space, with coordinates X, Y and Z, the field E is represented by the list of its three components (P,Q,R), where P=P(X,Y,Z), Q=Q(X,Y,Z) and R=R(X,Y,Z) are expressions in terms of the variables X, Y and Z and "variables" is the list { X Y Z }. Div(E) is therefore equal to:

$$\frac{\partial P}{\partial X} + \frac{\partial Q}{\partial Y} + \frac{\partial R}{\partial Z} .$$

Program 'DIVRG' is halted once the partial derivatives have been computed and a menu is displayed including the entries "DIVG", "EXIT" and one entry per variable (which enables us to enter values or purge them, etc.).

By pressing "DIVG", we can then calculate div(E) at specific points, or obtain it in symbolic form.

Press "EXIT" to quit the program.

'DIVRG': (Checksum: # 36444d, Size: 186.5 bytes)

```
«   DUP   PURGE   →   f   v
    «   0   1   f   SIZE   FOR   i
              f   i   GET   v   i   GET   ∂   +
        NEXT   'f'   STO
        {  {  "DIVG"  « f EVAL » } }   v   +
        {  {  "EXIT"   CONT  } }   +
        TMENU   HALT   v   PURGE   2   MENU
    »
»
```

PROGRAM 'DIVRG': PRACTICAL EXAMPLES

Example 1:

Calculate the divergence of the field `E(X,Y) = (COS(Y,X), EXP(X,Y)`, at any point in two-dimensional space (here, the point `P(X,Y)=COS(Y/X), W(X,Y)=EXP(XY) )`.

```
2: {'COS(Y/X)' 'EXP(X*Y)'}
1:         { X Y }
```

We first create the stack above and call 'DIVRG', which is halted after two seconds while a menu with the entries "DIVG", "X", "Y" and "EXIT" is displayed.

Pressing "DIVG" without giving a value for X and Y, we find the following at level 1 of the stack:

```
1:   '-(SIN(Y/X)*-(Y/X^2))+X*EXP(X*Y)'
```

Meaning that:

$$\mathrm{div}(E) = \frac{\partial P}{\partial X} + \frac{\partial Q}{\partial Y} = \frac{Y}{X^2}\,\mathrm{SIN}(\,Y\,/\,X\,) + X\,\mathrm{EXP}(XY).$$

We then press "EXIT" to quit the program.

Example 2:

We want to calculate the divergence of the field
$E(X,Y,Z) = (\,LN(X+Y),\ Y*Z^2,\ Z/(YZ)\,)$.
(Here, $P = LN(X+Y)$, $Q = Y*Z^2$ and $R = X/(YZ)$).
We first create the stack below and call 'DIVRG'.

```
2: { 'LN(X+Y)' 'Y*Z^2' 'X/Y/Z' }
1:          { X Y Z }
```

'DIVRG' is halted and a menu with the entries "DIVG", "X", "Y", "Z" and "EXIT" is displayed. We give X, Y and Z the values 1, 2 and 3 respectively (for example, enter 3 at level 1, press "Z", then STO).

Pressing "DIVG", we obtain a result immediately.
We find:

```
1:  9.27777777777
```

meaning that at the point (1,2,3):

$$\mathrm{Div}(E) = \frac{\partial P}{\partial X}(1,2,3) + \frac{\partial Q}{\partial Y}(1,2,3) + \frac{\partial R}{\partial Z}(1,2,3) = 167\,/\,18\ .$$

We then press "EXIT" to quit the program.

CURL OF A VECTOR FIELD

'CURL' calculates the curl curl(E) of a vector field E in three-dimensional space. The functional diagram is as follows:

2:	E	'CURL'	2:	
1:	variables	----------->	1:	curl(E)

The field E must be represented here by the list { P, W, R } of its components, each of which is an algebraic expression.
"variables" denotes the list of the three variables with respect to which the partial derivatives of P, Q and R are to be calculated.
Curl(E) is a list of 3 algebraic expressions representing the components of the curl of E, i.e. of the vector field (if P, Q and R are the components of E, and if X, Y and Z are the three variables):

$$\left(\frac{\partial R}{\partial Y} - \frac{\partial Q}{\partial Z} , \frac{\partial P}{\partial Z} - \frac{\partial R}{\partial X} , \frac{\partial Q}{\partial X} - \frac{\partial P}{\partial Y} \right)$$

N.B: Program 'CURL' is halted and a menu is displayed including the entries "CURL", "EXIT" (to quit) and one entry per variable (which enables us to enter values or purge them, etc.).

By pressing "CURL", we can thus obtain the curl expressed symbolically or its value at a point.

'CURL': (Checksum: # 27663d, Size: 327 bytes)

```
«  IF    DUP    SIZE    3    ==    THEN    DUP    PURGE    0    →    f   v   j
   «   1    3    FOR    i
            IF    i    3    ==    THEN    1    ELSE    i    1    +    END
            'j'    STO    f    j    GET    v    i    GET    ∂
            f    i    GET    v    j    GET    ∂    -
       NEXT
       3    ROLLD    3    →LIST    'f'    STO
       {    {    "CURL"
                 «    f    EVAL    1    3    START    ROT    EVAL    NEXT
                      IFERR    3    →ARRY    THEN    →LIST    END
                 »    }  }
       v    +    {  {  "EXIT"    CONT    }  }    +
       TMENU    HALT    v    PURGE    2    MENU
   »   END
»
```

Note:

If we calculate the curl at a specific point, the result is given in the form of a three-component vector.
Otherwise, the components of curl(E) are obtained in symbolic form and the result is given in the form of a list containing three elements.

PROGRAM 'CURL': PRACTICAL EXAMPLE

Example 1:

We want to calculate the curl of the vector field:

```
E(X,Y,Z) = ( XY, YZ, ZX )
```

We first create the stack below:

```
2:  { 'X*Y'  'Y*Z'  'Z*X'  }
1:        { X Y Z }
```

We then call program 'CURL'.

The program is halted and a menu displays the following entries:

"CURL", "X", "Y", "Z".

1) pressing "CURL" without giving values to X, Y and Z, we find:

```
1:  {  '-Z'  '-X'  '-Y'  }
```

The expression of the curl of E at any point `(X,Y,Z)` is therefore:

```
curl(E) = ( -Z, -X, -Y ).
```

2) pressing "CURL" after giving only X a value of 1, we find:

```
1:  {  '-Z'  -1  '-Y'  }
```

We thus obtain the expression of the curl of E at any point `(1,Y,Z)`.

3) pressing "CURL" after giving X, Y and Z the values 1, 2 and 3 respectively, we find:

```
1:  [  -3  -1  -2  ]
```

We thus obtain the value of the curl of E at the point `(1,2,3)`.

Press "EXIT" to quit the program.

GRADIENT OF A FUNCTION OF SEVERAL VARIABLES

'GRADI' calculates the gradient of a function f of several variables. The functional diagram is as follows:

```
   ┌─────────────┐                 ┌─────────────┐
2: │      f      │   'GRADI'    2: │             │
   ├─────────────┤  ----------->   ├─────────────┤
1: │  variables  │              1: │   grad(f)   │
   └─────────────┘                 └─────────────┘
```

Where "f" is an algebraic expression characterizing the function f.

"variables" is the list of variables with respect to which the partial derivatives are to be calculated.

"grad(f)" is the list of components of the vector of the gradient of f (where the components are expressed symbolically). If the user tells the calculator to compute the gradient for a specific point, the result is given as a vector.

For example, if f is a function of three variables X, Y and Z, then grad(f) is the vector:

$$\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right).$$

N.B: program 'GRADI' is halted to display a menu with the entries "GRADI", "EXIT" (to quit) and one entry per variable (which enables us to give each variable a value if required).

By pressing "GRADI", we can thus obtain the gradient expressed symbolically, or its value at a point.

'GRADI': (Checksum: # 27380d, Size: 235 bytes)

```
«  DUP   PURGE   DUP   SIZE   -   f   v   n
   «  1   n   FOR   i   f   v   i   GET   ∂   NEXT
      n   →LIST   'f'   STO
      {   {   "GRADI"
              «   1   n   FOR   i   f   i   GET   EVAL   NEXT
                  IFERR   n   →ARRY   THEN   →LIST   END
              »   }  }
      v   +
      {  {  "EXIT"   CONT   } }   +
      TMENU   HALT   v   PURGE   2   MENU
   »
»
```

PROGRAM 'GRADI': PRACTICAL EXAMPLES

Example 1:

We want to calculate the gradient of `f(X,Y) = EXP(XY)SIN(Y)`.
We first create the stack below:

```
2:  'EXP(X*Y)*SIN(Y)'
1:  { X Y }
```

We then call program 'GRADI'. The program is halted to display a personalized menu with the entries "GRADI", "X", "Y" and "EXIT".

1) pressing "GRADI" without giving values to X, Y and Z, we find (within 3 seconds):

```
1  { 'Y*EXP(X*Y)*SIN(Y)' 'X*EXP(X*Y)*SIN(Y)+EXP(X*Y)*COS(Y)'}
```

which is the expression of the gradient of f at any point (X,Y).

2) pressing "GRADI" after giving X and Y the values 1 and 2 respectively, we find (in 5 FIX mode):

```
1:  [ 13.43770 3.64392 ]
```

We thus obtain the value of the gradient of f at the point (1,2).
Press "EXIT" to quit the program.

Example 2:

We want to calculate the gradient of `f(X,Y,Z) = 'XY + Z² + Y/Z²'`.
We first create the stack below:

```
2:  'X*Y+Z^2+Y/Z'
1:  { X Y Z }
```

We then call program 'GRADI'. The program is halted to display a personalized menu with the entries "GRADI", "X", "Y" and "EXIT".

1) pressing "GRADI" without giving values to X, Y and Z, we find (within 2 seconds):

```
1:  {  Y 'X+1/Z' '2*Z-Y/Z^2' }
```

which is the expression of the grad(f) at any point (X,Y,Z).

2) pressing "GRADI" after giving X and Y the values 1 and 2 respectively, we find:

```
1:  {  2 '1+1/Z' '2*Z-2/Z^2' }
```

We thus obtain the expression the gradient of f at any point (1,2,Z).

LAPLACEAN OF A FUNCTION OF SEVERAL VARIABLES

'LAPL' calculates the Laplacean of a function f of several variables. The functional diagram is as follows:

```
   |              |                      |              |
   |--------------|                      |--------------|
2: |      f       |      'LAPL'      2:  |              |
   |--------------|   ------------->     |--------------|
1: |  variables   |                      |     △(f)     |
   ----------------                      ----------------
```

Where "f" is an algebraic expression characterizing the function f.

"variables" is the list of variables with respect to which the partial derivatives are to be calculated.

" △ (f)" is the expression of the Laplacean of f. If the user tells the calculator to compute for a specific point, the result is given as a real number.

For example, if f is a function of three variables x, y and z, then △ (f) is the scalar:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} .$$

N.B: program 'LAPL' is halted to display a menu with the entries "LAPL", "EXIT" (to quit) and one entry per variable (which enables us to give each variable a value if required).

By pressing "LAPL", we can thus obtain the Laplacean expressed symbolically, or its value at a point.

'LAPL': (Checksum: # 58922d, Size: 188.5 bytes)

```
«  DUP   PURGE   →   f   v
   «  0   1   v   SIZE   FOR   i
          f   1   2   START   v   i   GET   ∂   NEXT   +
      NEXT   'f'   STO
      {  {  "LAPL"   «  f  EVAL  »  }  }
      v   +
      {  {  "EXIT"   CONT   }  }   +
      TMENU   HALT   v   PURGE   2   MENU
   »
»
```

PROGRAM 'LAPL': PRACTICAL EXAMPLES

Example 1:

We want to calculate the Laplacean of `f(X,Y) = X^2*SIN(Y)`.
We first create the stack below:

```
2:  'X^2*SIN(Y)'
1:  { X Y }
```

We then call program 'LAPL'. The program is halted to display a personalized menu with the entries "LAPL", "X", "Y" and "EXIT".

1) pressing "LAPL" without giving values to `X` and `Y`, we find:

```
1:  '2*SIN(Y)+X^2*-SIN(Y)'
```

which is the expression of the Laplacean of f at a point `(X,Y)`.

2) pressing "LAPL" after giving X and Y the values 1 and 2 respectively, we find:

```
1:  909297426824
```

We thus obtain the value of the Laplacean of f at the point (1,2).
Press "EXIT" to quit the program.

Example 2:

We want to calculate the Laplacean of `f(X,Y,Z) = 'XY + Z² + Y/Z²'`.
We first create the stack below:

```
2:  'X*Y+Z^2+Y/Z'
1:  { X Y Z }
```

We then call program 'LAPL'. The program is halted to display a personalized menu with the entries "LAPL", "X", "Y", "Z" and "EXIT".

1) pressing "LAPL" without giving values to `X`, `Y` and `Z`, we find:

```
1:  '2+Y*(2*Z)/Z^2^2'
```

as the expression of `(F)` at a point `(X,Y,Z)` is:
`2 + 2*Y/Z^3` (use COLCT and EXPAN to see this).

2) pressing "LAPL" after giving `Z` a value of 1, we find the expression of the Laplacean of f at a point `(X,Y,1)`, i.e. `'2+Y*2'`.

Press "EXIT" to quit the program.

DIFFERENTIAL OF A FUNCTION OF SEVERAL VARIABLES

'DIFF' calculates the differential of a function f of several variables. The functional diagram is as follows:

```
   |             |                   |             |
   |-------------|                   |-------------|
2: |      f      |    'DIFF'     2:  |             |
   |-------------|  ----------->     |-------------|
1: |  variables  |                   |     df      |
   ---------------                   ---------------
```

Where "f" is an algebraic expression characterizing the function f.

"variables" is the list of variables with respect to which the partial derivatives are to be calculated.

"df" is the expression of the differential of f.

For example, if f is a function of three variables x, y and z, then df is written:

$$df = \frac{\partial f}{\partial X} dX + \frac{\partial f}{\partial X} dY + \frac{\partial f}{\partial X} dZ.$$

N.B: program 'DIFF' is halted to display a menu with the entries "DIFF", "EXIT" (to quit) and one entry per variable (which enables us to give each variable a value if required).

By pressing "DIFF", we can thus obtain the differential expressed symbolically, or its value at a point.

'DIFF': (Checksum: # 43931d, Size: 221 bytes)

```
«  DUP PURGE  →  f  v
   «  0  1  v  SIZE  FOR  i
         f  v  i  GET  ∂  "d"  v  i  GET  →STR
         2  OVER  SIZE  1  -  SUB  +  OBJ→  *  +
      NEXT
      'f'  STO
      { {  "DIFF"  « f  EVAL  »  }  }
      v  +
      { {  "EXIT"  CONT  }  }  +
      TMENU  HALT  v  PURGE  2  MENU
   »
»
```

N.B: If, for example, the variables are called X, Y and Z, program 'DIFF' uses the names 'dX', dY' and 'dZ'. For the program to run correctly, none of the variables must have the same name as another variable already in the directory.

PROGRAM 'DIFF': PRACTICAL EXAMPLES

Example 1:

We want to calculate the differential of `f(X,Y) = X^2*SIN(Y)`.
We first create the stack below:

```
2: | 'X^2*SIN(Y)' |
1: |   { X Y }    |
```

We then call program 'DIFF'. The program is halted to display a personalized menu with the entries "DIFF", "X", "Y" and "EXIT".

1) pressing "DIFF" without giving values to X and Y, we find:

```
1: | '2*X*SIN(Y)*dX+X^2*COS(Y)*dY' |
```

which is the expression of df at a point (X,Y).

2) pressing "DIFF" after giving X and Y the values 1 and 2 respectively, we find (in 5 FIX mode):

```
1: | '1.81859*dX-0.41615*dY' |
```

We thus obtain the value of df at the point (1,2).

Press "EXIT" to quit the program.

Example 2:

We want to calculate the differential of:
`f(RO,TETA,FI) = RO²*SIN(TETA)*EXP(FI*TETA).`

We first create the stack below:

```
2: | 'RO^2*SIN(TETA)*EXP(FI*TETA)' |
1: |       { RO TETA FI }          |
```

We then call 'DIFF'. The program is halted to display a personalized menu with the entries "DIFF", "RO", "TETA", "FI" and "EXIT".

If we give RO, TETA and FI the values 1, 2 and 3 respectively, then press "DIFF", we find (in 2 FIX mode):

```
1: | '733.67*dRO+932.62*dTETA+733.67*dFI' |
```

which is the expression of df at the point (1,2,3).

Press "EXIT" to quit the program.

GRAPHS

The HP48 has a graphic screen with a resolution of 131 x 64 pixels. A large number of instructions are related to managing this screen. These instructions are to be found in the **PLOT** or **PGR DSPL** menus, or in the **GRAPH** environment.

In terms of graphic display, the HP48 marks a considerable improvement on the HP28S, thus making the programs in this chapter, which were originally written for the HP28S, less essential than they were before.

However, I have kept the following programs, modifying them where necessary:

'PAR' : plotting a parametric curve (`X=X(T), Y=Y(T)`).

'POL' : plotting a curve with polar coordinates (`Ro = Ro(θ)`).

'POLP' : plotting a curve of a polar equation:
(`Ro = Ro(T), θ = θ(T)`).

'PLOT' : plotting program used by **'PAR'**, **'POL'** and **'POLP'**.

'ENV' : plotting an envelope of a family of straight lines.

'MTCL' : using the Monte-Carlo method to display curves of implicit functions: F(X,Y)=0.

'FAMT' : plotting a family of curves depending on T.

'ANIM' : producing successive screen images.

Programs 'PAR', 'POL', 'POLP' and 'ENV' are designed to plot curves while storing the points of the curve in the matrix ΣDAT, which is not possible using the DRAW instruction and proves useful if you need to plot curves on paper.

I thought twice about keeping program 'MTCL'. We can in fact display a curve `F(X,Y)=0` by plotting the expression `'F(X,Y)>0'` (or `'F(X,Y)<0'`,) using the DRAW instruction (first having defined "TRUTH" as the type of plot).

However, 'MTCL' can be quicker, as it can be used to test a part of the points on screen (and not all points as with "TRUTH" plots).

PLOTTING A PARAMETRIC CURVE

'PAR' enables you to plot a curve given by parametric equations: `X=X(T), Y=Y(T)`.

The functional diagram is as follows:

```
2: |            |    'PAR'      |        |
1: |    list    |   -------->   |  plot  |
```

The list at level 1 is in the format { M(T) start end step }, where:

* M(T) is an algebraic expression, whose value is the complex number giving the coordinates (X(t),Y(t)) of the point to be plotted (a capital T must be used here).
* "start" and "end" represent the initial value and the final value respectively of the parameter T.
* "step" is the increment of the parameter T. Plotting time is obviously inversely proportional to T.

Program 'PAR' calls program 'PLOT'.
Points to be noted:
Plotting is not stopped if a point cannot be evaluated. The point is simply left out.
Points found are put into the matrix ΣDAT (meaning that they can be re-read or used to change the screen display mode with the SCATRPLOT instruction).
Once a curve has been fully plotted, the calculator automatically goes into the 'GRAPH' environment.
Press 'ON' to quit the program

'PAR': (Checksum: # 35659d, Size: 37.5 bytes)

```
«  «  »  PLOT  »
```

Example:

Plot the curve `X(T)=6*COS(T), Y(T)=3*SIN(T)` using the default values in PAR, where `0 ≤ T ≤ 2*π` and a step of 0.2.

```
2: |                                          |  'PAR'    |        |
1: | { '6*COS(T)+i*3*SIN(T)' 0 '2*π' .2 }     |  ------>  |  plot  |
```

The curve plotted within 25 seconds is an ellipse.
Once the curve has been fully plotted, the 32 points used are stored in the matrix ΣDAT.

PLOTTING A CURVE WITH POLAR COORDINATES

'POL' enables you to plot a curve given by polar coordinates: Ro=Ro(T).
The functional diagram is as follows:

```
2: |            |      'POL'       |        |
1: |    list    |  ------------>   |  plot  |
```

The list at level 1 is in the format { Ro(T) start end step }, where:

* Ro(T) is an algebraic expression characterizing the radius vector of the point to be plotted, with a polar angle T (a capital T must be used here).

* "start" and "end" represent the initial value and the final value respectively of the polar angle T.

* "step" is the increment of the polar angle T. Plotting time is obviously inversely proportional to T.

Program 'POL' calls program 'PLOT'.
Points to be noted:
Plotting is not stopped if a point cannot be evaluated. The point is simply left out.
Points found are put into the matrix ΣDAT (meaning that they can be re-read or used to change the screen display mode with the SCATRPLOT instruction).
Once a curve has been fully plotted, the calculator automatically goes into the 'GRAPH' environment.
Press 'ON' to quit the program

'POL': (Checksum: # 48083d, Size: 52 bytes)

```
« « T i * EXP * » PLOT »
```

Example:

Plot the curve `Ro(θ) = 2 * (COS(2*θ) - 2*COS(θ))` using the default values in PAR, where `0 ≤ θ ≤ 2*π` and a step of 0.2.

```
2: |                                       |     'POL'     |        |
1: | { '2*(COS(2*T)-2*COS(T))' 0 '2*π' .2 } | ----------->  |  plot  |
```

The curve is obtained within 25 seconds.
Once the curve has been fully plotted, the 32 points used are stored in the matrix ΣDAT.

PLOTTING A CURVE OF A POLAR EQUATION

'POLP' enables you to plot a curve of a polar equation: `Ro=Ro(T), θ=θ(T)`.
The functional diagram is as follows:

```
2: |            |      'POLP'       |        |
1: |    list    |  ------------->   |  plot  |
```

The list at level 1 is in the format { M(T) start end step }, where:

* M(T) is an algebraic expression whose value is the complex number giving the pair of polar coordinates (Ro(T),θ(T)) of the point to be plotted (a capital T must be used here).
* "start" and "end" represent the initial value and the final value respectively of the parameter T.
* "step" is the increment of the parameter T. Plotting time is obviously inversely proportional to T.

Program 'POLP' calls program 'PLOT'.
Points to be noted:
Plotting is not stopped if a point cannot be evaluated. The point is simply left out.
Points found are put into the matrix ΣDAT (meaning that they can be re-read or used to change the screen display mode with the SCATRPLOT instruction).
Once a curve has been fully plotted, the calculator automatically goes into the 'GRAPH' environment.

Press 'ON' to quit the program

'POLP': (Checksum: # 35459d, Size: 51 bytes)

```
« « C→R i * EXP * » PLOT »
```

<u>Example</u>:

Plot the curve θ=SIN(Ro+1/Ro) using the default values in PAR, where .1 ≤ Ro ≤ 3 and a step of 0.1.

```
2: |                                  |   'POLP'      |        |
1: |  { 'T+i*SIN(T+1/T)' .1 3 .1 }    |  --------->   |  plot  |
```

The curve is obtained within 30 seconds.
Once the curve has been fully plotted, the 30 points used are stored in the matrix ΣDAT.

PLOTTING PROGRAM USED BY 'PAR', 'POL' and 'POLP'

Programs 'PAR', 'POL' and 'POLP' use the same 'PLOT' program, the text of which is shown below:

'PLOT': (Checksum: # 3577d, Size: 258.5 bytes)

```
«   SWAP    OBJ→    DROP    CLΣ
    ERASE    { #0  #0 }    PVIEW    DRAX    0
    →    sp    m    d    f    p    v
    «    d    →NUM    f    →NUM    FOR    i
                  i    'T'    STO
                  IFERR
                        m    →NUM    sp    →NUM
                        DUP    C→R    2    →ARRY    Σ+
                        DUP
                        IF    v    0    ≠
                              THEN    v    SWAP    LINE
                              ELSE    PIXON
                        END
                        'v'    STO
                  THEN
                        CLEAR
                  END
             p    STEP
             'T'    PURGE    GRAPH
    »
»
```

Points to be noted:

The variable 'v' contains the previous point (which is useful when plotting segments). We first put a value of 0 in 'o' and the test "IF v 0 ≠" checks that we are not on the first point (segments can only be plotted from the 2nd point onwards).

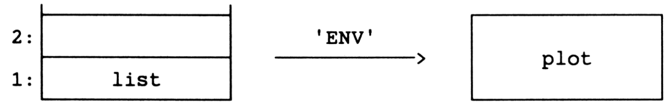
The various points (the end points of segments) are stored in the matrix **ΣDAT**, which is purged before the program is run.

PLOTTING THE ENVELOPE OF A FAMILY OF STRAIGHT LINES

'ENV' enables you to plot the envelope of a family of straight lines S(T), where the equation of S(T) is written:

`A(T)*X + B(T)*Y +C(T) = 0.`

The functional diagram is as follows:



The list at level 1 is in the format `{ A(T) B(T) C(T) start end step }`, where:

* A(T), B(T) and C(T) are algebraic expressions of the variable T (a capital T must be used here).
* "start" and "end" represent the initial value and the final value respectively of the parameter T.
* "step" is the increment of the parameter T. Plotting time is obviously inversely proportional to T.

Plotting is not stopped if an error occurs while computing a point on the curve. The point is simply left out.

Program 'ENV' calls program 'PAR', which calls program 'PLOT'.
The various points used to plot the curve are stored in the matrix ΣDAT .

'ENV': (Checksum: # 3853d, Size: 272.5 bytes)

```
«   'T'   PURGE   EVAL
    →   a   b   c   d   f   p
    «   a   'T'   ∂   b   'T'   ∂   c   'T'   ∂
        →   a1   b1   c1
        «   c1   b   *   c   b1   *   -   a   b1   *   a1   b   *
            -   DUP   ROT   ROT   /   SWAP   a1   c   *   a   c1
            *   -
    SWAP   /   i   *   +   d   f   p   4   →LIST   PAR
        »
    »
»
```

<u>Example</u>:

To plot the envelope of the family of straight lines:

`SIN(T)*XCOS(T)*Y-3*SIN(T)*COS(T)=0`, we put the following at level 1 of the list:

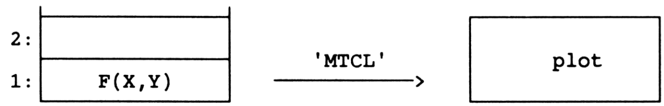
`{ 'SIN(T)' 'COS(T)' '-3*SIN(T)'*COS(T)' 0 '2*π' .1 }`

The curve obtained within 1 min 40 secs is an astroid.

MONTE CARLO METHOD (DISPLAYING A CURVE F(X,Y)=0)

We want to display a curve defined by the equation `F(X,Y)=0`. The principle used by the Monte Carlo method, implemented here by program 'MTCL', is to test the sign of F at a certain number of points (which are theoretically uniformly distributed) and to "highlight" the points where F is positive and leave those where F is negative.
We are thus able to display the two-dimensional areas delimited by the curve `F(X,Y)=0`, and thus get an idea of how the curve will look.

The functional diagram of 'MTCL' is as follows:



where `F(X,Y)` is the algebraic expression of the application F (X and Y must be capital).

Program 'MTCL' uses a "step" h. The default value of h is h=1.
h=1 means that all pixels are tested.
h=2 means that every other row and column are tested, etc.
Pressing the + key during plotting increases h by 1. Pressing the - key reduces h by 1 (but h must always remain between 1 and 8).

'MTCL': (Checksum: # 23081d, Size: 441.5 bytes)

```
«  1     →   f    h
   «     ERASE    { #0  #0 }    PVIEW    DRAX    PICT    SIZE
         PPAR    2    GET    PPAR    1    GET    DUP2
         -    C→R    5    ROLL    B→R    /    SWAP    5    ROLL    B→R    /
         →    pmax    pmin    sy    sx
         «    pmin    RE    pmax    RE    FOR    i    i    'X'    STO
                 pmin    IM    pmax    IM    FOR    j    j    'Y'    STO
                      f    EVAL
                      IF    0    >    THEN    i    j    R→C    PIXON    END
                      IF    KEY    THEN
                           { 85  95 }    SWAP    POS    1    +    { 0  -1  1 }
                           SWAP    GET    h    +    1    MAX    8    MIN    'h'    STO
                      END
                 sy    h    *    STEP
              sx    h    *    STEP
              { X  Y }    PURGE    GRAPH
         »
   »
»
```

Note:

Once the curve has been plotted, the calculator goes into the GRAPH environment. Press 'ON' to quit the environment.

PLOTTING A FAMILY OF CURVES VARYING ACCORDING TO A PARAMETER

==================================

'FAMT' lets you plot the various curves of a family of functions F varying according to a parameter t.
The functional diagram is as follows:

```
2: |          |     'FAMT'       |        |
1: |   list   |   ---------->    |  plot  |
```

the list at level 1 is in the format: { F(X,T) start end step }, where:

* F(X,T) is the algebraic expression characterizing the function F; X is the variable and T is the parameter (capitals must be used).
* "start" and "end" represent the initial value and the final value respectively of the parameter T.
* "step" is the increment of the parameter T.

First option:

Each curve is plotted on the screen one after the other (the screen clears between two plots). On quitting the program, a list containing the graphic objects for the plots is sent to level 1.

Second option:

Before calling 'FAMT', we enter the integer 1 at level 1 of the stack and the list of data therefore goes up to level 2.
All curves are then plotted one after the other, but this time all on screen at the same time. On quitting, a graphic object (representing the image of the screen display) is sent to level 1.

'FAMT': (Checksum: # 10702d, Size: 224 bytes)

```
« DUP 1 == DUP DROPN → t
  « OBJ→ DROP → d f p
    « FUNCTION 'X' INDEP ERASE DRAX STEQ
      IF t NOT THEN {} END
      d f FOR i
          i 'T' STO DRAW
          IF t NOT THEN
              PICT RCL + ERASE DRAX
          END
      p STEP
      IF t THEN PICT RCL END
      { EQ T } PURGE
    »
  »
»
```

Example:

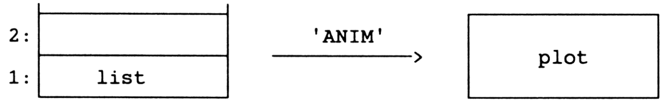
Using the default plotting parameters, plot `{ '2*SIN(X*T)*T' .5 1.5 .2 }`.

PRODUCING SUCCESSIVE SCREEN IMAGES

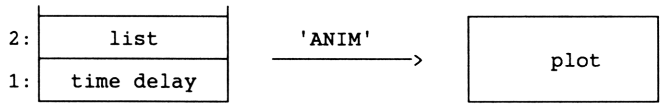
'ANIM' allows you to display different screen images successively.
These images must be represented by graphic objects grouped into a list.

The functional diagram is as follows:

First case:



Second case:



If the list has been stored in a variable, the name of that variable can be used. The images are then sent to the screen one after the other, in the order in which they appear in the list. When the last image has been displayed, the process loops back to the start.
Press the 'ON' key to quit program 'ANIM'.

In the first case:

The program halts when an image is displayed. 'ANIM' then waits for the user to press a key (any key except 'ON', used to quit the program) before displaying the next image.

In the second case:

"time delay" is a positive real number representing the delay, in seconds, between two successive displays. Images also scroll automatically on screen (the real number 0 takes you back to the first case).

'ANIM': (Checksum: # 57620d, Size: 99.5 bytes)

```
«  IF    DUP    TYPE    THEN    0    ELSE    ABS    END
   →     t
   «     { 1 }    { #0  #0 }    PVIEW
         DO
               GETI    PICT    { #0  #0 }    ROT    REPL    t    WAIT
               IF    t    NOT    THEN    DROP    END
         UNTIL    0    END
   »
»
```

LARGE INTEGERS

The programs in the 'LONG' directory are used to work with "large integers". By this, we mean positive or zero integers with a number of figures above the twelve significant figures that the calculator is capable of storing in memory.
Here, a large integer n is represented by a vector whose components represent the breakdown of n into base 10^5. These components must therefore be integers between 0 and 99999.

For example: `N = 165088696783290882115695` is written:
`[ 1650 88696 78329 8821 15695 ].`

Conversely:
`[ 87 132 6 78999 1 ]` represents `N = 8700132000067899900001.`

More simply, 1 is written `[ 1 ]` and `[ 1 0 ]` is equal to `100000 = 10^5.`

Here is the list of programs in the 'LONG' directory enabling you to perform standard operations on large integers:

'ADDL' : addition of two large integers.
'PRODL' : product of two large integers.
'POWL' : integer powers of a large integer.
'DIVL' : division of 2 large integers (calculation of quotient and integer remainders).
'GCDL' : gcd of two large integers.
'LCML' : lcm of two large integers.
'FACTL' : factorial of a natural integer, in large integer form.

The 'LONG' directory also includes routines enabling you to switch from one form of a large integer to another:

'L→ST' : writes a large integer as a string of characters.
'ST→L' : writes a string of characters in the form of a large integer.
'R→L' : switches from "real" to large integer form.
'L→R' : switches from large integer to "real" form.

The last two routines in the directory are designed to ensure that the programs listed above run correctly. The user should not usually have to use them directly:

'FRMT' : manages overflows in large integer calculations.
'ELML' : eliminates any zero coefficients to the left in a large integer.

Note:

Due to the way in which large integers and polynomials are represented, a certain number of large integer programs are in fact calls to similar polynomial programs (they therefore go into the 'POLY' directory before returning to the 'LONG' directory). Programs 'ADDL', 'PRODL' and 'ELML' make such calls.

ADDITION OF TWO LARGE INTEGERS

'ADDL' adds two large integers A and B, as shown in the functional diagram below:

```
2: |    A    |     'ADDL'      2: |         |
               ----------->
1: |    B    |                 1: |   A+B   |
```

The result is given in 'large integer' format. Program 'ADDL' goes into the 'POLY' directory, where it calls program 'ADDP'. It then returns to the 'LONG' directory, where it calls program 'FRMT' (overflow management).

'ADDL': (Checksum: # 63368d, Size: 48.5 bytes)

```
« POLY ADDP LONG FRMT »
```

Example: (in one second)

```
2: | [ 14 84721 632 ]      |   'ADDL'   2: |                           |
                             ------->
1: |[ 911 4839 95825 2]    |            1: |[ 911 4854 80546 634 ]     |
```

as `14 84721 00632 + 911 04839 95825 00002 = 911 04854 80546 00634 .`

PRODUCT OF TWO LARGE INTEGERS

'PRODL' multiplies two large integers A and B, as shown in the functional diagram below:

```
2: |    A    |     'PRODL'     2: |         |
               ----------->
1: |    B    |                 1: |   AB    |
```

The result is given in 'large integer' format. Program 'PRODL' goes into the 'POLY' directory, where it calls program 'PRODP'. It then returns to the 'LONG' directory, where it calls program 'FRMT' (overflow management).

'PRODL': (Checksum: # 40829d, Size: 50.5 bytes)

```
« POLY PRODP LONG FRMT »
```

Example: (in one second)

```
2: |   [ 14 84721 632 ]   |
                            --->
1: |[ 911 4839 95825 2]   |     1: |[13526 52696 63435 48708 30842 1264]|
```

as `14 84721 00632 * 911 04839 95825 00002 is equal to:`
`13526 52696 63435 48708 30842 01264.`

RAISING A LARGE INTEGER TO AN INTEGER POWER

'POWL' calculates A^n, where A is a large integer and n is a natural integer.
The functional diagram is as follows:

```
2:  |      A      |    'POWL'    2:  |             |
1:  |      n      |  ---------->  1:  |      A      |
```

N.B: 'POWL' calls program 'PRODL'.

'POWL': (Checksum: # 51318d, Size: 169 bytes)

```
«  →  a  n
   «  1  DUP  →ARRY
      WHILE  n  0  >
      REPEAT
         IF  n  2  MOD  THEN
               a  PRODL
         END
         n  2  /  FLOOR  'n'  STO
         IF  n  THEN
                  a  DUP  PRODL  'a'  STO
         END
      END
   »
»
```

Example: (within 3 seconds)

```
2: | [ 1 84721 632 ] |
1: |        3        | --> 1: |[6 30302 25019 87618 19996 4636 35968]|
```

as (1 84721 00632)^3 = 6 30302 25019 87618 19996 04636 35968.

DIVISION OF TWO LARGE INTEGERS

DIVL' divides two large integers A and B, as shown in the functional diagram below:

```
        |-----------------|                     |-----------------|
    2:  |        A        |       'DIVL'    2:  |        Q        |
        |-----------------|     ---------->     |-----------------|
    1:  |        B        |                 1:  |        R        |
        |-----------------|                     |-----------------|
```

where Q and R are the quotient and integer remainder respectively of the division (A = BQ + R, where R < B), both given in large integer format.

N.B: 'DIVL' calls programs 'ELML' and 'ADDL'.

'DIVL': (Checksum: # 27671d, Size: 411 bytes)

```
«  ELML   DUP   1   GET   1   +   OVER   SIZE   1   GET   0
   →  b   d   tb   q
   «  0   1   →ARRY   SWAP
      WHILE
         DUP   1   GET   d   /   FLOOR   'q'   STO
         DUP   SIZE   1   GET   tb   DUP2   >   3   ROLLD
         ==
         IF   DUP   q   NOT   AND   THEN
              3   PICK   b   -   ELML
              1   GET   0   ≥   'q'   STO
         END
         q   AND
         OR
      REPEAT
         IF   q   NOT   THEN
              OBJ→   1   GET   1   -   →ARRY   1   OVER
              1   GET   4   ROLL   100000   *   +
              DUP   d   /   FLOOR   'q'   STO   PUT
         END
         q   DUP   1   →ARRY   3   PICK   SIZE   1   GET
         tb   -   1   +   1   →LIST   RDM
         4   ROLL   ADDL   3   ROLLD
         b   *   NEG   OVER   SIZE   RDM   ADDL
      END
   »
»
```

Example: (within 9 seconds)

```
    |------------------------------------|                 |-----------------------|
 2: | [ 32415 738 2351 299 77314 ]       |   'DIVL'    2:  | [ 7 15359 17612 ]     |
    |------------------------------------|  ------->       |-----------------------|
 1: |      [ 4531 29119 77813 ]          |             1:  | [ 1887 12900 34758]   |
    |------------------------------------|                 |-----------------------|
```

as 32415 00738 02351 00299 77314 is equal to

(4531 29119 77813) * (7 15359 17612) + 1887 12900 34758.

GCD OF TWO LARGE INTEGERS

'GCDL' calculates the gcd (greatest common divisor) of two large integers A and B. The functional diagram is as follows:

```
   |           |                   |              |
2: |     A     |   'GCDL'       2: |              |
1: |     B     |  ---------->   1: | Pgcd( A,B )  |
```

The gcd is given in 'large integer' format.

N.B: 'GCDL' uses programs 'L→R', 'R→L', 'FRMT' and 'DIVL'. It also goes into the 'ARIT' directory , where it calls program 'GCD' (gcd of two integers).

'GCDL' also calls itself.

'GCDL': (Checksum: # 19457d, Size: 165.5 bytes)

```
«  IF   DUP   ABS   THEN
        IF   DUP2   SIZE   1   GET   SWAP   SIZE   1   GET   MAX   3   <
        THEN
             L→R   SWAP   L→R   ARIT   GCD   LONG   1   →ARRY   FRMT
        ELSE
             DUP   3   ROLLD   DIVL   SWAP   DROP   GCDL
        END
   ELSE
        DROP
   END
»
```

Example: (within 31 seconds)

```
  |                                |          |               |
2:| [ 116 14672  88165  91106 ]    | 'GCDL'   |               |
1:|  [ 4252 10326 53607 7700 ]     | ----> 1: |   [ 7858 ]    |
```

as the gcd of 116 14672 88165 91106 and of 4252 10326 53607 07700 is equal to 7858.

LCM OF TWO LARGE INTEGERS

'LCML' calculates the lcm (least common multiple) of two large integers A and B. The functional diagram is as follows:

```
2: |     A     |   'LCML'    2: |               |
               |  -------->     |               |
1: |     B     |             1: |  Ppcm( A,B )  |
```

The lcm is given in 'large integer' format.
N.B: 'LCML' calls programs 'GCDL', 'DIVL' and 'PRODL'.

'LCML': (Checksum: # 53667d, Size: 47 bytes)

```
« DUP2 GCDL DIVL DROP PRODL
»
```

Example: (within 20 seconds)

```
2: | [ 1245 865 ]   |  'LCML'   2: |                              |
                       ------>
1: | [ 7841 88500 ] |           1: | [ 19 52642 93146 10500 ]     |
```

as the csm of (1245 00865, 7841 88500) = 19 52642 93146 10500.

FACTORIAL

'FACTL' calculates the factorial n! of a natural integer n. The functional diagram is as follows:

```
                 'FACTL'
1: |    n    |  -------->   1: |    n!    |
```

n must be a natural integer. The result is given in 'large integer' form.
You can check for yourself that the "!" function on the HP48 only gives all the figures of n! up to n=14 (in which case n! = 87178291200). The purpose of 'FACTL' is to be able to obtain all the significant figures of n! when n>14.
Program 'FACTL' calls itself and 'FRMT'.

'FACTL': (Checksum: # 56148d, Size: 121 bytes)

```
« → n
  « IF n 14 ≤ THEN n ! 1 →ARRY FRMT
       ELSE n n 1 - FACTL * FRMT END
  »
»
```

Example:(in 4 seconds)

```
                 'FACTL'
1: |   25   |  -------->   1: | [ 1 55112 10043 33098 59840 0 ] |
```

as 25! = 1 55112 10043 33098 59840 00000.

WRITING A LARGE INTEGER AS A STRING OF CHARACTERS

'L→ST' transforms a large integer n and writes it as a string of characters representing n in spaced blocks of three figures.

The functional diagram is as follows:

```
     ┌─────────────┐    'L → ST'         ┌─────────────┐
  1: │      n      │   ──────────>   1:  │   string    │
     └─────────────┘                     └─────────────┘
```

'L→ST': (Checksum: # 46940d, Size: 233 bytes)

```
«   ELML   DUP   100000   CON   +   OBJ→   1   GET
    →   n
    «   " "   1   n   START
            SWAP   →STR   2   OVER   SIZE   SUB   SWAP   +
        NEXT
        WHILE   DUP   NUM   48   ==
            REPEAT   2   OVER   SIZE   SUB   END
        DUP   SIZE   3   -
        IF   DUP   0   >   THEN
            1   FOR   i
                DUP   1   i   SUB   " "   +   SWAP
                i   1   +   OVER   SIZE   SUB   +
            -3   STEP
        ELSE   DROP   END
    »
»
```

Example: (in one second)

```
[ 2456 818 54997 43411 ]   'L→ST'  ──────>   "2 456 008 185 499 743 411"
```

WRITING A STRING OF CHARACTERS AS A LARGE INTEGER

'ST→L' transforms a string of characters, consisting only of figures or spaces, and writes it as its corresponding large integer n.
The functional diagram is as follows:

```
1: | string |   'ST → L'   1: |   n   |
               --------->
```

The program is halted by an error message if no figure is found, or if an unauthorized character is encountered.

N.B: in the text below, the string "0123456789 " ends with a space.

'ST→L': (Checksum: # 16934d, Size: 364 bytes)

```
«   DUP   SIZE   DEPTH
    →   ch   n   p
    «   1   n   FOR   i
            "0123456789 "   ch   i   i   SUB   POS
            IF   DUP   THEN
                IF   DUP   11   ==   THEN   DROP
                ELSE   1   -   END
            ELSE   "Error"   DOERR   END
        NEXT
        DEPTH   p   -   2   +   DUP   5   MOD   DUP2   -
        →   n   r   m
        «   IF   r   THEN
                r   4   START   0   n   1   +   ROLLD   NEXT
            END
            m   1   +   m   5   /   1   +   FOR   k
                1   4   FOR   i
                    SWAP   10   i   ^   *   +
                NEXT
                k   ROLLD
            -4   STEP
            m   5   /   1   +   →ARRY
        »
    »
»
```

Example: (in 2 seconds)

```
1: | "5 178 895 411 702 658" |   'ST→L'   1: | [ 5 17889 54117 2658 ] |
                                ------->
```

WRITING A REAL NUMBER AS A LARGE INTEGER

'R→L' writes a real number N (usually an integer) as its equivalent 'large integer' NL.

The functional diagram is as follows:

```
1: [ N ]   --'R→L'-->   1: [ NL ]
```

N.B: 'R→L' calls program 'FRMT'.

'R→L': (Checksum: # 7262d, Size: 45.5 bytes)

```
« .5 + FLOOR 1 →ARRY FRMT »
```

Examples: 1 is written as [1].
71875488654 is written [7 18754 88654].
125487.7 is written [1 25488].
1.49865E30 is written [1 49865 0 0 0 0 0].

WRITING A LARGE INTEGER AS A REAL NUMBER

'L→R' writes a large integer NL as its real number equivalent N. The functional diagram is as follows:

```
1: [ NL ]   --'L→R'-->   1: [ N ]
```

Obviously, if NL is an integer greater than 1E12, then changing to real-number format will mean a loss in precision. Program 'L→R' is nevertheless useful if we want to find the order of magnitude of an integer written in 'large integer' form quickly.

N.B: program 'L→R' goes into the 'POLY' directory, where it calls program 'VALP'.

'L→R': (Checksum: # 34456d, Size: 50.5 bytes)

```
« POLY 100000 VALP LONG »
```

Example: (in under one second)

```
[ 48566 25781 59522 35678 ]   --'L→R'-->   1: 4.8566257816E19
```

MANAGING OVERFLOWS ON LARGE INTEGERS

When doing calculations on vectors representing large integers (e.g. a subtraction), the result may sometimes not exactly be in large integer format (with components all positive and less than 100,000). The purpose of 'FRMT' is to manage any such "overflows" and to put the vector into the required format.
'FRMT' is called by a large number of the programs in the 'LONG' directory. The user should not normally have to call it directly himself. The functional diagram is as follows:

```
   ┌─────────────┐    'FRMT'         ┌─────────────┐
1: │      V      │  ──────────>   1: │      W      │
   └─────────────┘                   └─────────────┘
```

where W is the resultant vector if V needs to be modified.

'FRMT': (Checksum: # 42172d, Size: 192.5 bytes)

```
«  0   SWAP   OBJ→   1   GET
   →   n
   «   1   n   START
           DUP   100000   MOD   DUP   n   3   +   ROLLD   -   100000   /   +
       NEXT
       WHILE   DUP   0   >
           REPEAT
               DUP   100000   MOD   DUP   'n'   1   STO+
               n   2   +   ROLLD   -   100000   /
           END
       DROP   n   →ARRY
   »
»
```

Examples:

`[ 106622 110050 129366 ]` is written `[ 1 6623 10051 29366 ].`
`[ 13902 -35682 -21383 ]` is written `[ 13901 64317 78617 ].`

ELIMINATING ZEROS TO THE LEFT

'ELML' eliminates all zero coefficients at the start of a vector. This routine is used by certain programs in the 'LONG' directory. It should not normally have to be called directly by the user. 'ELML' simply goes into the 'POLY' directory where it calls the program with the same name.

'ELML': (Checksum: # 11925d, Size: 41 bytes)

```
«   POLY   ELML   LONG   »
```

Example:

```
   ┌─────────────────┐    'ELML'         ┌─────────────────┐
1: │[ 0 0 157 67 ]   │  ──────────>   1: │  [ 157 67 ]     │
   └─────────────────┘                   └─────────────────┘
```

PROBABILITIES

The programs in the 'PROBA' directory enable you to find standard probability distributions and distribution functions (discrete or continuous) without having to use tables of numbers (and therefore without having to interpolate).
As enumeration is frequently required when calculating probabilities, the 'PROBA' directory also has a few small programs designed for this.

We should also note that certain other programs in other directories will be useful to us here. The most useful will be 'SIMP' and 'CALC' in the 'ARIT' directory and the instruction →Q (the value of a probability often has to be given as a simplified fraction rather than a real number). If, for example, we want to use 'CALC' from the 'PROBA' directory without transferring it to the 'ARIT' directory, we simply have to write the following program:

« ARIT CALC PROBA »

in the 'PROBA' directory (and call it 'CALC' again, although this is not absolutely necessary).

Here, then, is the list of programs in the 'PROBA' directory:

'CNP'	:	number of combinations without repetition of p objects selected from n.
'PNP'	:	number of permutations of p objects selected from n.
'GANP'	:	number of combinations with repetition of p objects selected from n.
'BINO'	:	list of binomial coefficients.
'BNP'	:	binomial distribution.
'BNPF'	:	binomial distribution function.
'HYP'	:	hypergeometric distribution.
'HYPF'	:	hypergeometric distribution function.
'POIS'	:	Poisson distribution.
'POISF'	:	Poisson distribution function.
'GEO'	:	geometric distribution.
'GEOF'	:	geometric distribution function.
'PRP'	:	Pascal distribution P(r,p).
'PRPF'	:	Pascal distribution function.
'JRP'	:	negative binomial distribution.
'JRPF'	:	negative binomial distribution function.
'EXPF'	:	exponential distribution function.
'→NRM'	:	normal distribution function.
'→SND'	:	standard normal distribution function.
'SND→'	:	inverse of the standard normal distribution function.
'NRM→'	:	inverse of the normal distribution function.
'FITN'	:	fitting a normal distribution N(m,σ).

COMBINATIONS WITHOUT REPETITION

'CNP' calculates the number of selections of p different objects from a set of n distinguishable objects (combinations without repetition). This number is denoted by C^p and is equal to:

$$\frac{n!}{p!(n-p)!}.$$

'CNP' simply operates the same way as COMB, but it is more easily accessible and more explicit. 'CNP' also gives a result of 0 if p is not between 0 and n.
The functional diagram is as follows:

		'CNP'		
2:	n	——>	2:	
1:	p		1:	C

'CNP': (Checksum: # 3917d, Size: 37.5 bytes)

```
« IFERR COMB THEN DROP2 0 END »
```

Example:

		'CNP'		
2:	25	——>	2:	
1:	8		1:	1081575

PERMUTATIONS

'PNP' calculates the number of permutations of p objects selected from a set of n distinguishable objects (ordered combinations without repetition). This number is denoted by P^p and is equal to:

$$\frac{n!}{(n-p)!}.$$

'PNP' simply operates the same way as PERM, but it is more easily accessible and more explicit. 'PNP' also gives a result of 0 if p is not between 0 and n.
The functional diagram is as follows:

		'PNP'		
2:	n	——>	2:	
1:	p		1:	A

'PNP': (Checksum: # 40761d, Size: 37.5 bytes)

```
« IFERR PERM THEN DROP2 0 END »
```

Example:

		'PNP'		
2:	25	——>	2:	
1:	8		1:	43609104000

COMBINATIONS WITH REPETITION

'GANP' calculates the number of combinations, with possible repetition, of p objects selected from a set of n distinguishable objects. This number is denoted by Γ_n^p and is equal to:

$$\Gamma_n^p = C_{n+p-1}^p$$

The functional diagram is as follows:

```
   |-----------|                  |-----------|
2: |     n     |    'GANP'     2: |           |
   |-----------|  ---------->     |-----------|
1: |     p     |               1: |     Γ     |
   |-----------|                  |-----------|
```

'GANP': (Checksum: # 7101d, Size: 40 bytes)

```
« DUP ROT + 1 - SWAP CNP »
```

Example:

```
   |-----------|                  |-----------|
2: |    25     |    'GANP'     2: |           |
   |-----------|  ---------->     |-----------|
1: |     8     |               1: | 10518300  |
   |-----------|                  |-----------|
```

LIST OF BINOMIAL COEFFICIENTS

'BINO' gives the list of binomial coefficients in the expansion of `(x+y)^n`, i.e. the coefficients:

C_n^k where $0 \le k \le n$ (list in order of k).

The functional diagram is as follows:

```
   |-----------|    'BINO'        |-----------|
1: |     n     |  ---------->  1: |   list    |
   |-----------|                  |-----------|
```

'BINO': (Checksum: # 23431d, Size: 70.5 bytes)

```
« → n
  « 0 n FOR i n i COMB NEXT
    n 1 + →LIST
  »
»
```

Example:

```
   |-------|    'BINO'        |----------------------------|
1: |   6   |  ---------->  1: | { 1 6 15 20 15 6 1 }       |
   |-------|                  |----------------------------|
```

BINOMIAL DISTRIBUTION B(n,p)

'BNP' calculates the probability Pr(X=k), where X is a discrete random variable that conforms to a binomial distribution with parameters n and p.

In other words, Pr(x=k) is the probability of obtaining k successes in a series of n independent trials at each of which the probability of success is p.

k and n must be integers where $0 \le k \le n$ and $0 \le p \le 1$.

The formula used is:

$$p(X=k) = C_n^k \; p\hat{}k \; (1-p)\hat{}(n-k).$$

The functional diagram is as follows:

3:	n		3:	
2:	p	'BNP' ——>	2:	
1:	k		1:	Pr(X = k)

N.B: 'BNP' calls program 'CNP'.

'BNP': (Checksum: # 2409d, Size: 109 bytes)

```
«  →   n   p   k
   «   n   k   CNP
       IF  DUP   THEN
           p   k   ^   1   p   -   n   k   -   ^   *   *
       END
   »
»
```

Example:

3:	25		3:	
2:	.5	'BNP' ——>	2:	
1:	10		1:	.097416639328

(we thus obtain the probability of obtaining heads 10 times if we toss a fair coin 25 consecutive times).

BINOMIAL DISTRIBUTION FUNCTION B(n,p)

'BNPF' calculates the probability `Pr(X ≤ k)`, where `X` is a discrete random variable that conforms to a binomial distribution with parameters n and p. We thus obtain the distribution function of `X`.

In other words, `Pr(X ≤ k)` is the probability of obtaining at most `k` successes in a series of n independent trials at each of which the probability of success is p.

`k` and `n` must be integers where `0 ≤ k ≤ n` and `0 ≤ p ≤ 1`.

The formula used is:

$$p(X \le k) = \sum_{i=0}^{i=k} C_n^i \; p^i \; (1-p)^{(n-i)}.$$

The functional diagram is as follows:

3:	n		3:	
2:	p	'BNPF' ——>	2:	
1:	k		1:	p(X ≤ k)

'BNPF': (Checksum: # 51659d, Size: 172.5 bytes)

```
«  →  n  p  k
   «  IF  n  2  /  k  <  THEN
   1  n  1  p  -  n  k  -  1  -  BNPF  -
      ELSE
         0
         0  k  FOR  j
               n  p  j  BNP  +
         NEXT
         k  0  ≥  *
      END
   »
»
```

Example: (within three seconds)

3:	100		3:	
2:	.166666666667	'BNPF' ——>	2:	
1:	15		1:	.387657551691

We thus obtain, for example, the probability of obtaining the result 6 at least 15 times if we throw a fair die 100 consecutive times.

HYPERGEOMETRIC DISTRIBUTION H(n,a,p)

'HYP' calculates the probability `Pr(X=k)`, where X is a discrete random variable that conforms to a hypergeometric distribution with parameters n, a and p.

To take an example, `Pr(X=k)` is the probability, when simultaneously selecting a different individuals from a total population n, of obtaining k individuals with a given property P, assuming that the proportion of individuals having that property in the total population is equal to p.

k, a and n must be integers where $0 \le k \le a \le n$ and $0 \le p \le 1$.

The formula used is:

$$p(X=k) = \frac{C_n^k \quad C_{n(1-p)}^{a-k}}{C_n^a}.$$

The functional diagram is as follows:

4:	n		4:	
3:	a		3:	
2:	p	'HYP'	2:	
1:	k	——>	1:	p(X = k)

'HYP': (Checksum: # 40776d, Size: 135.5 bytes)

```
«  →  n  a  p  k
   «  n  p  *  k  CNP
      n  1  p  -  *  a  k  -  CNP  *
      IF  DUP  THEN  n  a  CNP  /  END
   »
»
```

Example:

4:	60		4:	
3:	10		3:	
2:	.25	'HYP'	2:	
1:	3	——>	1:	.273864227268

We can say that this result represents the probability of obtaining exactly 3 white balls when drawing 10 balls without replacement out of a hat containing 60 balls, 15 of which are white (the proportion 15/60 = .25 is at level 2 of the stack).

HYPERGEOMETRIC DISTRIBUTION FUNCTION H(n,a,p)

'HYPF' calculates the probability `Pr(X ≤ k)`, where `X` is a discrete random variable that conforms to a hypergeometric distribution with parameters n, a and p. We thus obtain what is called the distribution function of `X`.
To take an example, `Pr(X≤k)` is the probability, when simultaneously selecting a different individuals from a total population n, of obtaining at most k individuals with a given property P, assuming that the proportion of individuals having that property in the total population is equal to p.

The functional diagram is as follows:

4:	n		4:	
3:	a		3:	
2:	p	'HYPF'	2:	
1:	k	——>	1:	p(X ≤ k)

'HYPF': (Checksum: # 39410d, Size: 221 bytes)

```
«  →  n  a  p  k
   «  'a-n*(1-p)'  EVAL  0  MAX  a  n  p  *  MIN
      →  mink  maxk
      «  IF  k  mink  ≥  k  maxk  ≤  AND  THEN
               0
               mink  k  FOR  i
                   n  a  p  i  HYP  +
               NEXT
         ELSE  0  END
      »
   »
»
```

Example: (in two seconds)

4:	30		4:	
3:	10		3:	
2:	.4	'HYPF'	2:	
1:	5	——>	1:	.881728366586

i.e.:
if we draw 10 balls without replacement out of a hat containing 30 balls (12 of which are white, the proportion of white balls therefore being .4), the probability of obtaining at most 5 is approximately 0.88.

POISSON DISTRIBUTION P(L)

'POIS' calculates the probability `Pr(X=k)`, where X is a discrete random variable that conforms to a Poisson distribution with parameter L. L is a strictly positive real number and k is a natural integer.

The formula used is:

$$p(X=k) = \exp(-L)\frac{L^k}{k!}$$

Poisson distributions can be used to construct models for solving traffic and queue problems.

The functional diagram is as follows:

```
2: [    L    ]    'POIS'      2: [           ]
1: [    k    ]  ---------->   1: [ p( X = k ) ]
```

'POIS': (Checksum: # 32314d, Size: 68 bytes)

```
« → L k 'EXP(-L)*L^k/k!'
»
```

Example:

```
2: [   2.5   ]    'POIS'
1: [   10    ]  ------->   1: [ 2.15725184496E-4 ]
```

POISSON DISTRIBUTION FUNCTION P(L)

'POISF' calculates the probability $Pr(X \le k)$, where X is a discrete random variable that conforms to a Poisson distribution with parameter L. L is a strictly positive real number and k is a natural integer.

The formula used is:

$$p(X \le k) = \exp(-L) \sum_{i=0}^{i=k} \frac{L^i}{i!}$$

Poisson distributions can be used to construct models for solving traffic and queue problems.

The functional diagram is as follows:

2:	L	'POISF'	2:	
1:	k	——————>	1:	p(X ≤ k)

'POISF': (Checksum: # 45358d, Size: 86.5 bytes)

```
«   →   L   k   ' Σ ( j = 0 , k , POIS ( L , j ) '
»
```

Example: (in under one second)

2:	5	'POISF'	2:	
1:	6	——————>	1:	.762183462973

GEOMETRIC DISTRIBUTION G(p)

'GEO' calculates the probability `Pr(X=k)`, where `X` is a discrete random variable that conforms to a geometric distribution with parameter p. p is a real number between 0 and 1 and k is a strictly positive integer.

The formula used is:

```
Pr(X=k) = p*(1-p)^(k-1)
```

The geometric distribution of the parameter p gives the number of failures before the first success in a series of independent trials at each of which the probability of success is p.

The functional diagram is as follows:

```
        |          |                      |           |
        |----------|                      |-----------|
   2:   |    p     |      'GEO'      2:   |           |
        |----------|  ------------>       |-----------|
   1:   |    k     |                 1:   | p( X = k )|
        ------------                      -------------
```

'GEO': (Checksum: # 63475d, Size: 74.5 bytes)

```
« → p k 'p*(1-p)^(k-1)*(k>0)'
»
```

Example:

```
        |          |                      |           |
        |----------|                      |-----------|
   2:   |   .25    |      'GEO'      2:   |           |
        |----------|  ------------>       |-----------|
   1:   |    3     |                 1:   |  .140625  |
        ------------                      -------------
```

In other words, if we draw cards out of a normal deck and replace them, the probability of the first club being the third card drawn is equal to 0.140625 (the probability of success at each draw is 0.25).

GEOMETRIC DISTRIBUTION FUNCTION G(p)

'GEOF' calculates the probability p(X≤k), where X is a discrete random variable that conforms to a geometric distribution with parameter p. p is a real number between 0 and 1 and k is a strictly positive integer.

The formula used is:

```
Pr(X≤k) = 1-(1-p)^k
```

The geometric distribution of the parameter p gives the number of failures before the first success in a series of independent trials at each of which the probability of success is p. We can therefore find the probability of the first success being obtained on or before the kth attempt.

The functional diagram is as follows:

2:	p	'GEOF' ———>	2:	
1:	k		1:	p(X ≤ k)

'GEOF': (Checksum: # 2016d, Size: 68.5 bytes)

```
«   →   p   k   '(1-(1-p)^k)*(k>0)'
»
```

Example:

2:	.5	'GEOF' ———>	2:	
1:	4		1:	.9375

In other words, if we toss a fair coin several times consecutively, the probability of obtaining heads on or before the 4th attempt is 0.9375.

PASCAL DISTRIBUTION P(r,p)

'PRP' calculates the probability `Pr(X=k)`, where X is a discrete random variable that conforms to a Pascal distribution with parameter r and p. p is a real number between 0 and 1 and r and k are both integers $(1 \le r \le k)$.

The formula used is:

$$p(X=k) = C_{k-1}^{r-1} \quad p\hat{\ }r*(1-p)\hat{\ }(k-r).$$

The Pascal distribution with parameters r and p gives the number of failures before the rth success in a series of independent trials at each of which the probability of success is p.
The geometric distribution with parameter p is quite simply the Pascal distribution with parameters 1 and p.

The functional diagram is as follows:

3:	r		3:	
2:	p	'PRP' ——>	2:	
1:	k		1:	p(X = k)

'PRP': (Checksum: # 13969d, Size: 126.5 bytes)

```
«  →  r  p  k
   «  k  1  -  r  1  -  CNP
      IF  DUP  THEN  'p^r*(1-p)^(k-r)'  EVAL  *  END
   »
»
```

Example:

3:	10		3:	
2:	.4	'PRP' ——>	2:	
1:	15		1:	.016323761406

If, for example, we draw a ball with replacement out of a hat containing 40% white balls, the probability of the 10th white ball being the 15th ball to be drawn is `0.016323761406`.

PASCAL DISTRIBUTION FUNCTION P(r,p)

'PRPF' calculates the probability $Pr(X \le k)$, where X is a discrete random variable that conforms to a Pascal distribution with parameter r and p. p is a real number between 0 and 1 and r and k are both integers $(1 \le r \le k)$.

The Pascal distribution with parameters r and p gives the number of failures before the rth success in a series of independent trials at each of which the probability of success is p. We can therefore find the probability of the rth success being obtained on or before at the kth attempt.

The functional diagram is as follows:

	Before			After
3:	r		3:	
2:	p	'PRPF'	2:	
1:	k	——————>	1:	p(X ≤ k)

'PRPF': (Checksum: # 53656d, Size: 95.5 bytes)

```
« → r p k '∑(j=r,k,PRP(r,p,j))'
»
```

Example:

	Before			After
3:	10		3:	
2:	.5	'PRPF'	2:	
1:	15	——————>	1:	.15087890625

If we toss a fair coin several times consecutively, the probability of the 10th "head" being obtained on or before the 15th attempt is 0.15087890625.

NEGATIVE BINOMIAL DISTRIBUTION J(r,p)

'JRP' calculates the probability `Pr(X=k)`, where `X` is a discrete random variable that conforms to a negative binomial distribution with parameters r and p. p is a real number between 0 and 1 and r and k are both integers $(1 \leq r,\ 0 \leq k)$.

The formula used is:

$$p(X=k)= C^{k}_{k+r-1}\quad p\hat{}r*(1-p)\hat{}k.$$

The negative binomial distribution with parameter r and p gives the number of failures before the rth success in a series of independent trials at each of which the probability of success is p.
Note:if we say that `X` conforms to the negative binomial distribution `J(r,p)` with parameters `n` and `p`, then `X+r` conforms to the Pascal distribution `P(r,p)` with parameters `r` and `p`.

The functional diagram is as follows:

Level	Input	'JRP' →	Output
3:	r		
2:	p		
1:	k		p(X = k)

N.B: program 'JRP' calls program 'PRP'.

'JRP': (Checksum: # 32211d, Size: 31.5 bytes)

```
« 3 PICK + PRP »
```

Example:

Level	Input	'JRP' →	Output
3:	5		
2:	.25		
1:	20		3.29075176397E-2

If we draw cards with replacement from a normal deck of cards, the probability of drawing exactly 20 cards that are not clubs before drawing the fifth club is approximately equal to 0.0329.

NEGATIVE BINOMIAL DISTRIBUTION FUNCTION J(r,p)

'JRPF' calculates the probability `Pr(X ≤ k)`, where `X` is a discrete random variable that conforms to a negative binomial distribution with parameters r and p. p is a real number between 0 and 1 and r and k are both integers `(1 ≤ r, 0 ≤ k)`.

The negative binomial distribution with parameter r and p gives the number of failures before the rth success in a series of independent trials at each of which the probability of success is p.

We can therefore find the probability of the number of failures before the rth success being equal at most to k.

The functional diagram is as follows:

3:	r		3:	
2:	p	'JRPF' ——>	2:	
1:	k		1:	p(X ≤ k)

N.B: program 'JRPF' calls program 'PRPF'.

'JRPF': (Checksum: # 23661d, Size: 33.5 bytes)

```
« 3 PICK + PRPF »
```

Example:

3:	5		3:	
2:	.6	'JRPF' ——>	2:	
1:	4		1:	.73343232

If we draw a ball with replacement out of a hat containing 60% white balls, the probability of the number of non-white balls drawn before the 5th white ball being equal to 4 at the most is `0.73343232`.

EXPONENTIAL DISTRIBUTION FUNCTION

'EXPF' gives the probability `Pr(X ≤ x)`, where `X` is a discrete random variable that conforms to an exponential distribution with parameter L, and x is a given real number.

The formula used is: $p(X \leq x) = 0$ if $x \leq 0$.

and if $x \geq 0$: $p(X \leq x) = \int_0^x L \exp(-Lt)\, dt = 1 - \exp(-Lx)$

The functional diagram is as follows:

	Stack			Stack
2:	L	'EXPF' →	2:	
1:	x		1:	p(X ≤ x)

'EXPF': (Checksum: # 6632d, Size: 68.5 bytes)

```
« → L x '(1-EXP(-L*x))*(x≥0)'
»
```

Example:

	Stack			Stack
2:	2	'EXPF' →	2:	
1:	1		1:	.864664716763

NORMAL DISTRIBUTION FUNCTION N(m,σ)

'→NRM' calculates the probability `Pr(X ≤ x)`, where `X` is a discrete random variable that conforms to a normal distribution with parameters m and σ. m and σ are both real numbers (m represents the expectation of `X` and σ is the standard deviation of `X`. Obviously, $\sigma > 0$). x is any real number. The formula used is:

$$p(X \le x) = \frac{1}{\sigma\sqrt{(2\pi)}} \int_{-\infty}^{x} \exp(\ -(t-m)^2/(2\sigma^2)\ )\ dt$$

Normal distributions are useful in constructing models of problems related to large populations. The functional diagram is as follows:

Level	Input	'→ NRM'	Level	Output
3:	m		3:	
2:	σ	——>	2:	
1:	x		1:	p(X ≤ x)

'→NRM': (Checksum: # 64285d, Size: 36 bytes)

```
«  SWAP  SQ  SWAP  UTPN  NEG  1  +  »
```

Example:

Level	Input	'→ NRM'	Level	Output
3:	4.5		3:	
2:	.4	——>	2:	
1:	5		1:	.894350226333

STANDARD NORMAL DISTRIBUTION FUNCTION

'→SND' calculates `Pr(X ≤ x)`, where `X` is a discrete random variable that conforms to a standard normal distribution `N(0, 1)`, i.e. a special case of a normal distribution where `m=0` and `σ=1`.
The functional diagram below is therefore simpler:

Level	Input	'→ SND'	Level	Output
1:	x	——>	1:	p(X ≤ x)

N.B: program '→SND' calls program '→NRM'.

'→SND': (Checksum: # 52295d, Size: 33.5 bytes)

```
«  0  1  ROT  →NRM  »
```

Example:

Level	Input	'→ SND'	Level	Output
1:	.71	——>	1:	.76114793191

INVERSE OF THE NORMAL DISTRIBUTION FUNCTION N(0,1)

'SND→' calculates the inverse of the standard normal distribution function N(0,1). If X is a discrete random variable conforming to this distribution and p is a number between 0 and 1, we calculate the unique real value of x such that p(X≤x)=p.
The functional diagram is as follows:

```
   ┌───────────┐     'SND→'       ┌───────────┐
1: │     p     │   ──────────>  1: │     x     │
   └───────────┘                  └───────────┘
```

Note:'SND→' uses the ROOT instruction to solve the equation p(X≤x)=p. It is absolutely necessary that p be an element within the interval]0,1[.

'SND→': (Checksum: # 53511d, Size: 93.5 bytes)

```
«   →   p
    «   «   1   p   -   0   1   X   UTPN   -   »
    'X'   0   ROOT   'X'   PURGE
    »
»
```

INVERSE OF THE NORMAL DISTRIBUTION FUNCTION N(m,σ)

'NRM→' calculates the inverse of the normal distribution function N(m,σ). If X is a discrete random variable conforming to this distribution and p is a number between 0 and 1, we calculate the unique real value of x such that p(X≤x)=p.
The functional diagram is as follows:

```
   ┌───────────┐                  ┌───────────┐
3: │     m     │               3: │           │
   ├───────────┤                  ├───────────┤
2: │     σ     │    'NRM→'     2: │           │
   ├───────────┤   ──────────>    ├───────────┤
1: │     p     │               1: │     x     │
   └───────────┘                  └───────────┘
```

Note:'NRM→' calls program 'SND→'. It is absolutely necessary that p be an element within the interval]0,1[.

'NRM→': (Checksum: # 35961d, Size: 31 bytes)

```
«   SND→   *   +   »
```

Example: to calculate the 3rd quartile of the distribution N(2, 1.5), we enter:
2 1.5 .75 NRM→ to obtain 3.01173462529 in 3 seconds.

FITTING A NORMAL DISTRIBUTION N(m,σ)

'FITN' enables you to calculate the best possible fit of a normal distribution `N(m,σ)` to a distribution for which at least two values of the distribution function are known (or to a simple statistic for which at least two points on the cumulative frequency polygon are known).

We therefore assume that, for a random variable `X`, a certain number of points: `(a,b), (c,d), ..., (x,y)` are known such that:

`p(X≤a)=b, p(X≤c)=d, ..., p(X≤x)=y,`

i.e. a certain number of values of the distribution function of `X`.

Program 'FITN' calculates the mean (m) and the standard deviation (σ) of the normal distribution `N(m,σ)` that is the best fit to the distribution of the variable `X`.

Before calling 'FITN', you should enter your data in the matrix ΣDAT as follows, using the notation given above:

```
        [[ a b ]
ΣDAT =   [ c d ]
        .......
         [ x y ]]
```

The order of the rows in ΣDAT is not important.

The functional diagram is as follows:

```
                                    2: | m: ..... |
            'FITN'
1: |     |  ------->                1: | σ: ..... |
```

'FITN': (Checksum: # 50027d, Size: 117 bytes)

```
«  ΣDAT   DUP
   1   NΣ   FOR   i
       i   { 2 }   +   DUP2   GET   SND   PUT
   NEXT
   STOΣ   2   1   COLΣ   LR   ROT   STOΣ
   SWAP   DTAG   "m"   →TAG   SWAP   DTAG   "σ"   →TAG
»
```

N.B: on quitting 'FITN', the contents of ΣDAT remain unchanged. 'FITN' calls 'SND→'.
We must be careful to make sure that all the elements in the second column are between 0 and 1.

PROGRAM 'FITN': PRACTICAL EXAMPLES

Example 1:

A random variable X conforms to a normal distribution. We know that:
Pr(X≤1) = .1635 and Pr(X≤5) = .8053.
Calculate the mean m and the standard deviation σ of X.
Here, there is an exact solution to the problem, which can be found using 'FITN'.

We put the matrix

```
[[ 1  .1635 ]
 [ 5  .8053 ]]
```

in ΣDAT, and call 'FITN'.

Within nine seconds, we obtain (in 4 FIX mode):

2:	m: 3.1298
1:	σ: 2.1729

Therefore, X conforms to a normal distribution N(m,σ)
with a mean m = 3.1298
and standard deviation σ = 2.1729.
We can check these results with the sequence 1 →NRM, which gives the result 0.1635 from this stack.

Example 2:

Let us take a population of 100 individuals classified according to their height in cm:

Height	Population
[150,160[	6
[160,165[	11
[165,170[	20
[170,175[	25
[175,180[	20
[180,185[	11
[185,190[	5
[190,200[	2

We want to estimate the variable H equal to the height of a random individual taken from this population using a normal distribution.
We first calculate the vector of cumulative absolute frequencies [6 17 37 .. 100].
We then divide by the total number of observations (100) to obtain the vector of the cumulative relative frequencies: [0.06 0.17 0.37 0.62 0.82 0.93 0.98 1].
We put the following matrix in ΣDAT:

```
[[ 160 0.06 ]
 [ 165 0.17 ]
 [ 170 0.37 ]
 [ 175 0.37 ]
 [ 180 0.82 ]
 [ 185 0.93 ]
 [ 190 0.98 ]]
```

(Note that we leave out the point (200,1) to avoid 'FITN' locking out).
We then call program 'FITN'.
Within 37 seconds, we obtain (in 2 FIX mode):

2:	m: 172.75
1:	σ: 8.26

The height h of an individual in the population described can therefore best be approximated by the normal distribution N(172.75,8.26).

SIMPLE STATISTICS

The programs in the 'STAT1' directory are for studying simple discrete or class statistics with known frequencies (absolute or relative) corresponding to each value or each class of an attribute X to be studied.

For a discrete statistic `(Xi, Ni)`, where `Xi` represents the values of the attribute `X` and `Ni` the frequencies (absolute or relative), we put pairs of `(Xi, Ni)` in the matrix ΣDAT (`Xi` values in the first column, Ni values in the second).

For a class statistic `([Ai,A(i+1)[ ,Ni)`, we put a list that itself consists of sub-lists in a variable called DATA. The first sub-list contains the class limits and the second the frequencies (absolute or relative).

For example, the statistic:

Xi	[0,2[	[2,6[	[6,8[	[8,14[
Ni	22	25	15	10

will be represented in the variable DATA by the following list:

```
{  { 0 2 6 8 14 }  { 22 25 15 10 }  }.
```

A variable called ΣBAK is used in certain programs in the 'STAT1' directory to back up the contents of ΣDAT or to show intermediate calculations leading up to a given result (calculation of the Gini coefficient, for example).

Here is the list of programs in the 'STAT1' directory:

- **C→D** : transformation of a class statistic into a discrete statistic.
- **MK** : calculating the kth moment.
- **KMM** : calculating the kth moment about the mean.
- **SKMM** : calculating the standard kth moment about the mean.
- **YULE** : Yule, Kelley and Pearson coefficients.
- **QTLE** : calculating quantiles.
- **MDEV** : calculating the mean deviation.
- **GEOM** : calculating the geometric mean.
- **HARM** : calculating the harmonic mean.
- **MDL** : calculating the value equal to 50% of the cumulative mass (computation table in ΣBAK).
- **GINI** : calculating the Gini coefficient (computation table in ΣBAK).
- **LRTZ** : plotting a Lorentz curve (or concentration curve).
- **HIST** : plotting a histogram.
- **CFP** : plotting a cumulative frequency polygon.
- **→CUM** : creating a cumulative table.
- **CUM→** : going back from a cumulative table to an initial table.
- **COL.N** : displays any column in ΣBAK (for GINI and MDL).
- **C.COL** : calculates a column depending on 2 columns in ΣDAT.
- **MODΣ** : modifying one or two columns in ΣDAT.

TRANSFORMING A CLASS STATISTIC INTO A DISCRETE STATISTIC

'C→D' transforms a simple class statistic into a discrete statistic. This is done by concentrating the absolute class frequency at the mean.

The initial grouped statistic must be in 'DATA'. The result is sent to ΣDAT.

The stack is left unchanged by 'C→D'. However, an error message will appear when calling 'C→D' if the first sub-list in 'DATA' does not contain exactly one more element than the second.

'C→D': (Checksum: # 15895d, Size: 111 bytes)

```
«   DATA   1   GET   OBJ→
    →  n
    «  2   n   START
               OVER   +   2   /   n   ROLLD
           NEXT
           DROP   DATA   2   GET   OBJ→
           { 2 }  SWAP   +   →ARRY   TRN   STOΣ
    »
»
```

Example:

The statistic:

Xi	[0,2[	[2,6[	[6,8[	[8,14[
Ni	22	25	15	10

,

represented in the variable DATA by the list:

```
{     { 0 2 6 8 14 }     { 22 25 15 10 }     }.
```

is transformed into the matrix:

```
[ [  1 22   ]
  [  4 25   ]
  [  7 15   ]
  [  11 10  ] ]
```

This matrix is then put into ΣDAT.

KTH MOMENT OF A SIMPLE STATISTIC

'MK' calculates the kth moment (denoted by Mk) of the discrete statistic in the variable ΣDAT. When dealing with a class statistic (in the variable 'DATA'), we first have to call program 'C→D' to transform it into a discrete statistic.

The formula giving the kth moment of the statistic (Xi, Ni) is:

$$M_k = \frac{1}{N} \Sigma \; Ni \; Xi^k \quad \text{(N = total number of observations = } \Sigma \text{ Ni)..}$$

If k=1, we obtain the arithmetic mean (Σ Ni Xi) / N.
If k=2, we obtain the square (Σ Ni Xi^2) / N of the quadratic mean.
The functional diagram is as follows:

```
         ┌──────────┐     'MK'          ┌──────────┐
    1:   │    k     │  ──────────>  1:  │    Mk    │
         └──────────┘                   └──────────┘
```

'MK':(Checksum: # 17889d, Size: 80.5 bytes)

```
«  →   k
    «   ΣDAT   OBJ→   DROP   0
        1   NΣ   START
            ROT   k   ^   ROT   *   +
        NEXT
        TOT   2   GET   /
        »
    »
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT by the two-column matrix:

```
[ [ 1  22 ]
  [ 4  25 ]
  [ 7  15 ]
  [ 11 10 ] ],
```

we obtain, for example (in under one second):

```
         ┌──────────┐     'MK'          ┌────────────────┐
    1:   │    1     │  ──────────>  1:  │ 4.68055555556  │
         └──────────┘                   └────────────────┘
```

(The arithmetic mean is therefore approximately 4.68).

```
         ┌──────────┐     'MK'          ┌──────────┐
    1:   │    2     │  ──────────>  1:  │  32.875  │
         └──────────┘                   └──────────┘
```

(The quadratic mean is therefore $\sqrt{32.875} \approx 5.73$).

KTH MOMENT ABOUT THE MEAN OF A SIMPLE STATISTIC

'KMM' calculates the kth moment about the mean (denoted by μk) of the discrete statistic in the variable ΣDAT. When dealing with a class statistic (in the variable 'DATA'), we first have to call program 'C→D' to transform it into a discrete statistic.

The formula giving the kth moment about the mean of the statistic (Xi, Ni) is:

$$\mu_k = \frac{1}{N} \Sigma \text{ Ni } (\text{Xi}-\overline{\text{X}})^k \qquad (\text{N} = \Sigma \text{ Ni}, \ \overline{\text{X}} = \text{mean}).$$

If k=1, we obtain 0.
If k=2, we obtain the variance (Σ Ni $(\text{Xi}-\overline{\text{X}})^2$ / N (i.e. the square of the standard deviation).
The functional diagram is as follows:

1:	k	'KMM' ——>	1:	µk

N.B: Program 'KMM' calls program 'MK'.

'KMM':(Checksum: # 54860d, Size: 108.5 bytes)

```
«  →   k
   «   ΣDAT   OBJ→   DROP   1   MK
       →   m
       «   0   1   NΣ   START
                ROT   m   -   k   ^   ROT   *   +
           NEXT
           TOT   2   GET   /
       »
   »
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT by the two-column matrix:

```
[ [ 1   22 ]
  [ 4   25 ]
  [ 7   15 ]
  [ 11  10 ] ],
```

we obtain, for example (in one second):

1:	2	'KMM' ——>	1:	10.9673996914

(The variance is therefore approximately 10.97 and the standard deviation $\sqrt{(\mu 2)} \approx 3.31$).

1:	3	'KMM' ——>	1:	22.3069112224

KTH STANDARD MOMENT ABOUT THE MEAN OF A SIMPLE STATISTIC

======================================

'SKMM' calculates the kth standard moment about the mean (denoted by αk) of the discrete statistic in the variable ΣDAT. When dealing with a class statistic (in the variable 'DATA'), we first have to call program 'C→D' to transform it into a discrete statistic.

The formula giving the kth standard moment about the mean of the statistic `(Xi, Ni)` is:

```
ak = µk / (σ^k)
```

where μk is the kth moment about the mean (see program 'KMM') and σ is the standard deviation.

If k=1, we obtain 0.
If k=2, we obtain 1.
If k=3, we obtain the coefficient of skewness.
If k=4, we obtain the coefficient of kurtosis.

The functional diagram is as follows:

```
1: [ k ]   'SKMM' ------>   1: [ αk ]
```

N.B: Program 'SKMM' calls program 'KMM'.

'SKMM':(Checksum: # 10141d, Size: 46.5 bytes)

```
« DUP KMM 2 KMM √ ROT ^ /
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT
by the two-column matrix:

```
[ [ 1  22 ]
  [ 4  25 ]
  [ 7  15 ]
  [ 11 10 ] ],
```

we obtain, for example (in under two seconds):

```
1: [ 3 ]   'SKMM' ------>   1: [ .61416340071 ]

1: [ 4 ]   'SKMM' ------>   1: [ 2.35842660849 ]
```

YULE, KELLEY AND PEARSON COEFFICIENTS

'YULE' calculates the Yule, Kelley and Pearson coefficients of a class statistic in the variable 'DATA'.

The formulae used are:

Yule's coefficient: `s = (q3+q1-2M) / (q3-q1).`

Kelley's coefficient: `k = 2*(q3-q1) / (d9-d1).`

Pearson's 1st coefficient: `p1 = 3*(m-M) / σ.`
Pearson's 2nd coefficient: `p2 = (m-md) / σ` (for a unimodal statistical series with mode md).

Where `q1` and `q3` are the first and third quartiles.
`d1` and `d9` are the first and ninth deciles.
`m` is the arithmetic mean.
`M` is the median σ is the standard deviation.

<u>Note</u>:

Pearson's 2nd coefficient is only useful for classes with the same amplitude. If the statistic is not unimodal, this coefficient is not displayed.

The functional diagram is as follows:

4:			YULE= ....
3:			KELLEY= ....
2:		'YULE'	PEARSON1= ....
1:		——>	PEARSON2= ....

<u>N.B</u>: 'YULE' calls programs 'QTLE', 'C→D', 'MK' and 'KMM'.

TEXT OF PROGRAM 'YULE' AND PRACTICAL EXAMPLE

'YULE':(Checksum: # 49682d, Size: 535.5 bytes)

```
«   { .1  .25  .5  .75  .9 }    QTLE    EVAL    C D
    1   MK    2    KMM    √    MAXΣ    2    GET    0    0
    →   d1    q1    med    q3    d9    moy    sig    max    num    lig
    «    1    NΣ    FOR    i
             'ΣDAT(i,2)'    EVAL
             IF    max    ==    THEN
                  'num' 1    STO+    i    'lig'    STO
             END
         NEXT
         q3    q1    +    med    2    *    -
         q3    q1    -    /    "YULE"    →TAG
         q3    q1    -    d9    d1    -
         /    2    *    "KELLEY"    →TAG
         moy    med    -    sig    /    3    *
         "PEARSON1"    →TAG
         IF    num    1    ==    THEN
         moy    'ΣDAT(lig,1)'    EVAL    -    sig    /
         "PEARSON2"    →TAG
         END
    »
»
```

Example:

If we take the statistic:

Xi	[0,5[	[5,10[	[10,15[	[15,20[
Ni	22	25	15	10

represented in the variable 'DATA' by the list:

{ { 0 5 10 15 20 } { 22 25 15 10 } },

we call 'YULE', which gives us the following within 4 seconds:

4:	YULE: 9.99999999953E-2
3:	KELLEY: 1.11658456486
2:	PEARSON1: .355182651996
1:	PEARSON2: .177318528263

QUANTILES OF A SIMPLE STATISTIC

'QTLE' calculates the quantiles of the grouped (class) statistic in the variable 'DATA'.

Note:

If q is a real number between 0 and 1, its corresponding quantile is the value of the attribute X for which the proportion of the cumulative absolute frequency is equal to q. The quantile is calculated by linear interpolation in the class encompassing this proportion.

For example:

If `q=.5`, we obtain the median of the statistic (value of the attribute at which half the absolute frequency is reached).

If `q=k/10` (`k` integer, $1 \leq k \leq 9$), we obtain the kth decile dk.

If `q=k/4` (k integer, $1 \leq k \leq 3$), we obtain the kth quartile qk. Deciles, centiles and 1,000-iles, etc. are defined in the same way.

The functional diagram is as follows:

```
    |--------------|     'QTLE'         |--------------|
 1: |     list     |  ------------>  1: |  quantiles   |
    |______________|                    |______________|
```

where "list" is the list of proportions q from `]0,1[` for which we want to find the quantiles, and "quantiles" is the corresponding list of quantiles.

Examples:

list = `{ .5 }` if we only want the median.
list = `{ .25 .75 }` if we want the first and third quartile.

N.B:

If a proportion q in "list" is not between 0 and 1, the corresponding quantile (which therefore cannot exist) is replaced in the list obtained by the string of characters representing q.

TEXT OF PROGRAM 'QTLE' AND PRACTICAL EXAMPLE

'QTLE':(Checksum: # 20526d, Size: 243 bytes)

```
«  0   DATA   2   GET   +   0   +  →CUM   DUP   DUP   SIZE   GET
   →   e   k
   «   1   OVER   SIZE   FOR   j
          j   DUP2   GET
          IF   DUP   FLOOR   THEN
               →STR
          ELSE
               k   *   e   2
               WHILE   GETI   4   PICK   ≤   REPEAT   END
               2   -   DUP   3   ROLLD   GETI   3   ROLLD
               GET   OVER   5   ROLL   -   3   ROLLD   -   /
               DATA   1   GET   ROT   GETI   3   ROLLD
               GET   OVER   -   ROT   *   +
          END
          PUT
       NEXT
   »
»
```

Example:

If we take the statistic:

Xi	[0,2[	[2,6[	[6,8[	[8,14[
Ni	22	25	15	10

which will be represented in the variable 'DATA' by the list:

`{ { 0 2 6 8 14 } { 22 25 15 10 } },`

we put `{ .1 .25 .5 .75 .9 }` at level 1 and then call 'QTLE'.

Within 2 to 3 seconds we obtain the following list at level 1 of the stack (in 3 FIX mode):

`{ 0.655 1.636 4.240 6.933 9.680 }.`

Meaning, for example, that:
the first decile d1 ≈ .655
the median M ≈ 4.240 and the third quartile ≈ 6.933.

MEAN DEVIATION OF A SIMPLE STATISTIC

'MDEV' calculates the mean deviation of the discrete statistic in the variable ΣDAT. When dealing with a class statistic (in the variable 'DATA'), we first have to call program 'C→D' to transform it into a discrete statistic.

The formula giving the mean deviation of the statistic `(Xi, Ni)` is:

$$\frac{1}{N} \Sigma \; Ni \; | \; Xi - \overline{X} \; |$$

where N is the total number of observations ΣNi and $\overline{X}$ is the arithmetic mean.

The functional diagram is as follows:

```
1: [                ]   'MDEV'  ------>   1: [mean deviation]
```

N.B: Program 'MDEV' calls program 'MK'.

'MDEV': (Checksum: # 54392d, Size: 93 bytes)

```
«   ΣDAT   OBJ→   DROP   1   MK
    →   m
    «   0   1   NΣ   START
              ROT   m   -   ABS   ROT   *   +
        NEXT
        TOT   2   GET   /
    »
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT by the two-column matrix:

```
[ [ 1   22 ]
  [ 4   25 ]
  [ 7   15 ]
  [ 11  10 ] ],
```

we obtain, for example (in one second):

```
1: [                ]   'MDEV'  ------>   1: [ 2.72183641975 ]
```

GEOMETRIC MEAN OF A SIMPLE STATISTIC

'GEOM' calculates the geometric mean Mg of the discrete statistic in the variable ΣDAT. When dealing with a class statistic (in the variable 'DATA'), we first have to call program 'C→D' to transform it into a discrete statistic.

The formula giving the geometric mean of the statistic `(Xi, Ni)` is:

$$\left[\prod (Xi)^{\wedge}(Ni) \right]^{\wedge}(1/N)$$

where N is the total number of observations frequency ΣNi.

The functional diagram is as follows:

```
1: [            ]  'GEOM'  ---->  1: [     Mg     ]
```

'GEOM': (Checksum: # 12712d, Size: 80 bytes)

```
«  ΣDAT   OBJ→   DROP   TOT   2   GET
   →  t
   «  1   1   NΣ   START
              3   ROLLD   t   /   ^   *
         NEXT
   »
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT by the two-column matrix:

```
[ [ 1  22 ]
  [ 4  25 ]
  [ 7  15 ]
  [ 11 10 ] ],
```

we obtain, for example (in under one second):

```
1: [            ]  'GEOM'  ---->  1: [ 3.38648044939 ]
```

HARMONIC MEAN OF A SIMPLE STATISTIC

'HARM' calculates the harmonic mean Mh of the discrete statistic in the variable ΣDAT. When dealing with a class statistic (in the variable 'DATA'), we first have to call program 'C→D' to transform it into a discrete statistic.

The formula giving the harmonic mean of the statistic (Xi, Ni) is:

$$\frac{1}{Mh} = \frac{1}{N} \Sigma \frac{Ni}{Xi}$$

where N is the total number of observations ΣNi.

The functional diagram is as follows:

```
1: [            ]     'HARM'      1: [     Mh     ]
                  ------------->
```

'HARM': (Checksum: # 20323d, Size: 82.5 bytes)

```
«  ΣDAT   OBJ→   DROP   TOT   2   GET
   →   t
   «   0   1   NΣ   START
            SWAP   ROT   /   t   /   +
       NEXT
       INV
   »
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT by the two-column matrix:

```
[ [ 1  22 ]
  [ 4  25 ]
  [ 7  15 ]
  [ 11 10 ] ],
```

we obtain, for example (in under one second):

```
1: [            ]     'HARM'      1: [ 2.30017633025 ]
                  ------------->
```

MEDIAL OF A SIMPLE STATISTIC

'MDL' calculates the value equal to 50% of the cumulative mass of the grouped (class) statistic X in the variable 'DATA'.

Note:
This value is the value of the attribute X for which we are able to reach 50% of the cumulative "masses" of the attribute.
If the statistic X is given by the pairs `([Ai,A(i+I) [,Ni)`, the mass corresponding to the class `[Ai,A(i+I)[` with an absolute frequency Ni is:

```
Ni*( A(i+I) + A(i) )/2.
```

The value is then calculated by linear interpolation in the class encompassing 50% of the cumulative mass.

Program 'MDL' puts the table required to calculated the value equal to 50% of the cumulative mass in 'ΣBAK'.

Row N°i of this table has the following form (there are p rows for p classes):

class centre	absolute frequency	mass	cumulative mass
Xi	Ni	NiXi	$\sum_{j \le i} NjXj$

The functional diagram is as follows:

```
                    'MDL'
1: |            |  ------->  1: |     M1     |
```

N.B: Program 'MDL' calls program 'C→D'.

TEXT OF PROGRAM 'MDL' AND PRACTICAL EXAMPLE

'MDL':(Checksum: # 28648d, Size: 272 bytes)

```
«  C→D    ΣX*Y    0
   →   m    md
   «   0    1    NΣ   FOR i
           ΣDAT    i    { 1 }    +    GETI    3    ROLLD    GET
           DUP2    *    4    PICK    OVER    +
           IF   5   PICK   m   2   /   <   OVER   m   2   /   ≥   AND
           THEN
                DUP   m   2   /   -   3   PICK   /   DATA   1   GET
                i   GETI   3   ROLLD   GET   DUP   ROT   -   ROT   *   -
                'md'   STO
           END
       NEXT
       NΣ    { 4 }    +    →ARRY
       'ΣBAK'    STO    DROP    md
   »
»
```

Example:

If we take the statistic:

Xi	[0,2[	[2,6[	[6,8[	[8,14[
Ni	22	25	15	10

represented in the variable DATA by the list:

{ { 0 2 6 8 14 } { 22 25 15 10 } }

we obtain in two seconds:

1: [ ] —'MDL'→ 1: 6.88571428571

GINI COEFFICIENT OF A SIMPLE STATISTIC

'GINI' calculates the Gini coefficient (or concentration coefficient) of the discrete statistic in the variable ΣDAT. When dealing with a class statistic (in the variable 'DATA'), we first have to call program 'C→D' to transform it into a discrete statistic.

This number is always between 0 and 1. The more concentrated the statistic is, the more it approaches 1 (i.e. the masses of the statistical distribution are mostly spread over a relatively small number of individuals).

The intermediate calculations required to compute the Gini coefficient (and also to plot the Lorentz curve, see program 'LRTZ') are stored in a table backed up in the variable 'ΣBAK'. Row N°i of this table has the following form (there are as many rows as there are values of the attribute studied):

column1	column2	column3	column4	column5	column6
class centre	absolute frequency	cumulative absolute frequency	% Ui of cum. abs.freq.	mass	cumulative mass
Xi	Ni	$\sum_{j\le i} N_j$	$\frac{100}{N}\sum_{j\le i} N_j$	NiXi	$\sum_{j\le i} N_jX_j$

column7	column8	column9	column10
% Vi of cum. mass.	base of trapezium	height of trapezium	area of trapezium
$\frac{100}{M}\sum_{j\le i} N_jX_j$	Ui-U(i-1)	$\frac{Vi+V(i+1)}{2}$	base * height

The Gini coefficient is equal to 1 - (total area)/5,000, where "total area" is the sum of coefficients in the tenth column of 'ΣBAK'.

The functional diagram of 'GINI' is as follows:

```
1: [            ]   'GINI'  ---->   1: [ coefficient ]
```

TEXT OF PROGRAM 'GINI' AND PRACTICAL EXAMPLE

'GINI':(Checksum: # 50275d, Size: 335.5 bytes)

```
«   TOT  2    GET    ΣX*Y    0
    →    n    m      a
    «    0    0    0    0    0    0    0    0
         1    NΣ    FOR    i
              ΣDAT    i    {1}    +    GETI    3    ROLLD    GET    DUP2    *
              OVER    12    PICK    +    DUP    100    *    n    /    ROT    DUP
              11    PICK    +    DUP    100    *    m    /    4    PICK    15    PICK
              -    OVER    13    PICK    +    2    /    DUP2    *    DUP
              'a'    STO+
         NEXT
         NΣ    {10}    +    →ARRY    'ΣBAK'    STO    8    DROPN
         1    a    5000    /    -
    »
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT
by the two-column matrix:

```
[ [ 1  22 ]
  [ 4  25 ]
  [ 7  15 ]
  [ 11 10 ] ],
```

we obtain, for example (in 2 seconds):

```
1: [            ]   'GINI'  ------>   1: [ .383489943952 ]
```

LORENTZ CURVE OF A SIMPLE STATISTIC

'LRTZ' plots the Lorentz curve (or concentration or Gini curve) of the discrete statistic in the variable ΣDAT. When dealing with a class statistic (in the variable 'DATA'), we first have to call program 'C→D' to transform it into a discrete statistic.

Important note:

For program 'LRTZ' to run properly, we first have to run program 'GINI' (because we use the contents of the table 'ΣBAK').

Using the notation from program 'GINI', the Lorentz curve is the polygonal line within the square `[0,100]x[0,100]` and joining the point `(0,0)` to the point `(100,100)` passing through the points (Ui,Vi). It is located underneath the first bisector.
The plotted curve fills the whole of the HP48's screen, giving it a horizontal scale of "131 pixels = 100 units" and a vertical scale of "64 pixels = 100 units". This breaks with the convention of displaying a Gini curve within a square, but ensures greater legibility on the screen.
N.B: the vertical sides of the trapezium are plotted.

The coordinates Ui and Vi of the points plotted are in columns 4 and 7 of 'ΣBAK'.
When the curve has been plotted, we go into the graphic environment GRAPH. The stack can be displayed by pressing "ON".

N.B: Program 'LRTZ':
- * Goes into the 'MATR' directory in order to use 'GETC'.
- * Then returns to the 'STAT1' directory.

'LRTZ':(Checksum: # 57409d, Size: 280.5 bytes)

```
«  (0,0)   ΣBAK   MATR   DUP   4   GETC   SWAP   7   GETC   STAT1
   R→C   OBJ→   1   GET   →LIST   +
   0   100   XRNG   0   100   YRNG   ERASE
   { #0d  #0d }   PVIEW   DRAX
   1   OVER   SIZE   1   -   FOR   i
       DUP   i   1   +   GET   OVER   i   GET   OVER   LINE
       DUP   RE   0   R→C   SWAP   LINE
   NEXT   DROP
   (0,0)   (100,100)   LINE   GRAPH
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT by the two-column matrix:

```
[ [ 1   22 ]
  [ 4   25 ]
  [ 7   15 ]
  [ 11  10 ] ],
```

the curve is plotted within 4 seconds.

HISTOGRAM OF A SIMPLE STATISTIC

'HIST' plots the histogram of a grouped (class) statistic X in the variable 'DATA'.

The histogram is plotted in such a way that it takes up as much screen space as possible.

Once it has been plotted, we go into the graphic environment GRAPH. We can display the stack by pressing "ON".

'HIST':(Checksum: # 40551d, Size: 249 bytes)

```
«  DATA    EVAL
   1   OVER    SIZE    FOR    i
       i    DUP2    GET    4    PICK    i    GETI
       3    ROLLD    GET    SWAP    -    /    PUT
   NEXT
   OVER    1    GET    0    R→C    PMIN    OVER
   DUP    SIZE    GET    OVER    OBJ→
   2    SWAP    START    MAX    NEXT
   R→C    PMAX    ERASE    { #0    #0 }    PVIEW    DRAX
   0    +    1    OVER    SIZE    1    -    FOR    i
       DUP    i    GET    3    PICK
       i    GETI    3    ROLLD    GET
       0    R→C    3    ROLLD    SWAP    R→C    BOX
   NEXT
   DROP2    GRAPH
»
```

Example:

If we take the statistic:

Xi	[0,2[	[2,6[	[6,8[	[8,14[
Ni	22	25	15	10

which is represented in the variable 'DATA' by the list:

`{ { 0 2 6 8 14 } { 22 25 15 10 } },`

the histogram is plotted within 4 seconds.

CUMULATIVE FREQUENCY POLYGON OF A SIMPLE STATISTIC

'CFP' plots the cumulative frequency polygon of a grouped (class) statistic X in the variable 'DATA'.

The polygon is plotted in such a way that it takes up as much screen space as possible.

Once it has been plotted, we go into the graphic environment GRAPH. We can display the stack by pressing "ON".

N.B: Program 'CFP' uses program '→CUM'.

'CFP':(Checksum: # 56583d, Size: 204.5 bytes)

```
«  DATA    EVAL   CUM    OVER    1    GET    0    R→C    PMIN
   1   2    START    OVER    DUP    SIZE    GET    NEXT
   R→C    PMAX    ERASE    { #0  #0 }    PVIEW
   DRAX    SWAP    OBJ→    →ARRY
   0    ROT    +    OBJ→    →ARRY    R→C
   2    OVER    SIZE    1    GET    FOR    i
        DUP    i    GET    OVER    i    1    -    GET    OVER    LINE
        DUP    RE    0    R→C    LINE
   NEXT
   DROP    GRAPH
»
```

Example:

If we take the statistic:

Xi	[0,2[	[2,6[	[6,8[	[8,14[
Ni	22	25	15	10

which is represented in the variable 'DATA' by the list:

`{  { 0 2 6 8 14 }  { 22 25 15 10 } },`

the polygon is plotted within 4 seconds.

CREATING A CUMULATIVE TABLE

'→CUM' creates a table B of the same format as an initial table A by accumulating the elements in table A with a lower index. A may be a vector or a matrix with one or more columns. Totals are given in the natural order of coefficients (from left to right along a row and from one row to the next row below).

Being able to create such tables is very useful for studying simple statistics.

The functional diagram is as follows:

```
      |-----------|      ' CUM'          |-----------|
   1: |     A     |   ------------>   1: |     B     |
```

Note:'→CUM' is also able to calculate the totals of elements in a list.

'→CUM': (Checksum: # 20274d, Size: 92 bytes)

```
«  2    OVER    SIZE    { }    +    OBJ→
   IF    2    ==    THEN    *    END
   FOR    i
        i    DUP2    1    -    GET    3    PICK    i    GET    +    PUT
   NEXT
»
```

Examples:

```
      |-----------------|     ' CUM'          |-----------------|
   1: | [ 1 2 3 4 5 ]   |  ------------>   1: |[ 1 3 6 10 15]   |
```

```
   |                       |                 |                       |
   |-----------------------|                 |-----------------------|
   |[[ 99.7362347512 ]     |                 |[[ 99.7362347512 ]     |
   |  [ 741.839762611 ]    |                 |  [ 841.575997362 ]    |
   |  [ 1078.75865479 ]    |    ' CUM'       |  [ 1920.33465215 ]    |
1: |  [ 1162.21562809 ]]   |  --------->  1: |  [ 3082.55028024 ]]   |
```

```
   |                  |                  |                  |
   |------------------|                  |------------------|
   |[[  1  2  3 ]     |     ' CUM'       |[[ 1   3   6 ]    |
1: |  [  4  5  6 ]]   |  --------->  1:  |  [ 10 15  21 ]]  |
```

GOING BACK FROM A CUMULATIVE TABLE TO AN INITIAL TABLE

'CUM→' lets you "break down" a cumulative table and thus do the reverse of what you do with '→CUM'.

Program '→CUM' allows you to create a cumulative table, whereas program 'CUM→' allows you to go back to the initial table you started with.

The functional diagram is as follows:

```
1: [ A ]   'CUM→'   1: [ B ]
          ------->
```

Here A is the initial table (matrix or vector) and B is the final table (A therefore represents the cumulative table of B).

Note:'CUM→' is also able to "break down" a list.

'CUM→': (Checksum: # 10763d, Size: 94.5 bytes)

```
«  DUP   SIZE   {}   +   OBJ→
   IF   2   ==   THEN   *   END
   2   FOR   i
      i   DUP2   GET   3   PICK   i   1   -   GET   -   PUT
   -1   STEP
»
```

Examples:

```
1: [ 1 3 6 10 15 ]   'CUM→'   1: [ 1 2 3 4 5 ]
                    ------->
```

```
   [[ 99.7362347512 ]              [[ 99.7362347512 ]
    [ 841.575997362 ]               [ 741.839762611 ]
    [ 1920.33465215 ]    'CUM→'     [ 1078.75865479 ]
1:  [ 3082.55028024 ]]  ------->  1: [ 1162.21562809 ]]
```

```
   [[ 1  3  6 ]     'CUM→'      [[ 1  2  3 ]
1:  [ 10 15 21 ]]  ------->  1:  [ 4  5  6 ]]
```

EXTRACTING A COLUMN FROM ΣBAK

'COL.N' extracts column number n from the matrix 'ΣBAK' in the 'STAT1' directory.
The result is given in the form of a column matrix C. We also obtain the sum Σ of terms in the column extracted at level 1 of the stack.

The functional diagram is as follows:

	Before			After
2:		'COLN' →	2:	C
1:	n		1:	Σ

N.B: Program 'COL.N' calls program 'GETC' in the 'MATR' directory.

Program 'COL.N' is meant to be used along with programs 'MDL' and 'GINI'. When calculating the medial and the Gini coefficient of a discrete statistic using these two programs, we in fact also have to load the table of intermediate calculations into the variable 'ΣBAK'. It can therefore be very useful to be able to consult this table column by column and to find the sum of the columns.

'COL.N': (Checksum: # 23519d, Size: 75.5 bytes)

```
« ΣBAK SWAP MATR GETC STAT1 DUP OBJ→
  1 GET 2 SWAP START + NEXT
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT by the two-column matrix:

```
[ [ 1  22 ]
  [ 4  25 ]
  [ 7  15 ]
  [ 11 10 ] ],
```

and call program 'GINI', the table of intermediate calculations is put into 'ΣBAK'. We can thus obtain (in 2 FIX mode):

	Before			After
2:		'C.COL.' →	2:	[[99.74] [741.84] [1078.76] [1162.22]]
1:	10		1:	3082.55

We thus find the value of the areas of the trapezia used to construct the Gini curve. The Gini coefficient is computed immediately afterwards:

$$I \approx 1 - 3082.55/5000 \approx 0.38$$

CREATING A COLUMN AS A FUNCTION OF COLUMNS IN ΣDAT

The matrix ΣDAT must contain the two columns representing a discrete statistic (the first for the values of the attribute, or for the class centres, and the second for the absolute frequencies).
'C.COL' lets you calculate a column as a function of the two columns in ΣDAT.

The functional diagram is as follows:

```
2: |           |   'C.COL'    2: |    C    |
   |-----------|  --------->     |---------|
1: |  f (X, N) |              1: |    Σ    |
```

where f(X,N) is the function used (an algebraic expression or program) and X and N denote the elements of the first and second column of ΣDAT respectively (capitals must be used), and C is the column matrix obtained, where Σ is the sum of coefficients in the column C.
Program 'C.COL' is very useful for studying simple statistics. It allows you, for example, to create a table of moments of a statistic (variance, etc.) column by column and within a very short time.

'C.COL':(Checksum: # 3810d, Size: 176 bytes)

```
«  0   →   p   s
   «   p   ΣDAT   OBJ→   DROP
       NΣ  2  *  1  -  NΣ  FOR  i
           'N'  STO  'X'  STO  p  EVAL  DUP
           's'  STO+  i  ROLLD
       -1  STEP
       NΣ  { 1 }  +  →ARRY  SWAP  DROP  s
   »
   { X N }  PURGE
»
```

Example:

If we take the statistic:

Xi	1	4	7	11
Ni	22	25	15	10

represented in ΣDAT by the two-column matrix:

```
[ [ 1   22 ]
  [ 4   25 ]
  [ 7   15 ]
  [ 11  10 ] ],
```

We obtain, for example, within two seconds:

```
2: |            |             2: | [[    22 ]  |
   |            |   'C.COL'      |  [  1600 ]  |
   |            |  --------->    |  [  5145 ]  |
   |            |                |  [ 13310 ]] |
   |------------|                |-------------|
1: | 'N*X^3'    |             1: |    20077    |
```

MODIFYING COLUMNS IN ΣDAT

The matrix ΣDAT must theoretically contain the two columns representing a discrete statistic (the first for the values of the attribute, or for the class centres, and the second for the frequencies). 'MODΣ' lets you modify either column (or both) in ΣDAT.

The functional diagram is as follows:

```
        ┌────────────┐      'MODΣ'          ┌────────────┐
     1: │    list    │   ──────────>     1: │            │
        └────────────┘                      └────────────┘
```

where "list" is a list in the format { prog1 prog2 } where:

> prog1 is the program applied to the first column.
> prog2 is the program applied to the second column.

prog1 and prog2 must be written in Reverse Polish Notation and must give a numerical result as a function of the element in the stack. If we wish to keep the first column unchanged, we simply write prog1 = « ».

If prog2 is not in the list the second column remains unchanged.
The contents of ΣDAT are backed up and stored in 'ΣBAK'.

'MODΣ':(Checksum: # 50513d, Size: 152.5 bytes)

```
«   ΣDAT    'ΣBAK'    STO    OBJ→
    →   n
    «   'ΣDAT'
        1    NΣ    FOR   j
             j    { 1 }    +
             n    1    FOR   i
                  DUP2    GET    3    i    +    PICK    EVAL    PUTI
             -1   STEP
             DROP
        NEXT
        n    1    +    DROPN
    »
»
```

Example:

If we take the statistic:

Xi	55	65	75	85
Ni	220	250	150	100

represented in ΣDAT by the two-column matrix:

```
[ [ 55 220 ]
  [ 65 250 ]
  [ 75 150 ]
  [ 85 100 ] ],
```

We enter `{ « 5 / 14 - » « 10 / » }` at level 1 of the stack and then call 'MODΣ'.

Within 2 to 3 seconds, the matrix ΣDAT shows:

```
[ [ -3  22 ]
  [ -1  25 ]
  [  1  15 ]
  [  3  10 ] ]
```

and the previous contents of ΣDAT are stored in ΣBAK.

Note:'MODΣ' is useful for changing the scale and/or origin.

STATISTICS IN TWO VARIABLES

The programs in the 'STAT2' directory can be used to study discrete statistics `in two variables (X,Y)`, where the absolute or relative frequency of each value `(Xi,Yi)` is known. This frequency (absolute or relative) is denoted by `Nij`.

The different values of `X` (the first marginal statistic of the pair `(X,Y)`) must be entered, in vector form, in a variable called '`X`'.

Likewise, the different values of `Y` (the second marginal statistic of the pair `(X,Y)`) must be entered, in vector form, in a variable called '`Y`'.

The frequency table with the absolute frequencies Nij must be entered in a variable 'ΣFRQ', in matrix form, according to the following conventions:

The statistic X is therefore in the first column. The values of Y are in the first row of the table. The matrix 'ΣFRQ' contains a table with n rows and p columns with a general term `Nij`, which is the absolute frequency of the value `(Xi,Yj)` of the pair `(X,Y)`.

	Y1	Y2	Y3 ......Yp
X1	N1,1	N1,2	N1,3 ... N1,p
X2	N2,1	N2,2	N2,3 ... N2,p
..	....		
Xn	Nn,1	Nn,2	Nn,3 ... Nn,p

<u>Note</u>:

We will often want to study a statistic in two variables `(X,Y)` consisting of a set of points `(Xi,Yi)` with an absolute frequency of 1, i.e. in the following form:

X	X1	X2	X3	.....	Xn
Y	Y1	Y2	Y3	.....	Yn

Certain programs in the 'STAT2' directory are well suited to this simple example. However, the programs that use 'ΣFRQ' may still be used if the nth order identity matrix is put in 'ΣFRQ'.

Here is the list of programs in the 'STAT2' directory:

'FRQX' : frequencies of the first marginal statistic.

'FRQY' : frequencies of the second marginal statistic.

'MX' : arithmetic mean of the first marginal statistic.

'MY' : arithmetic mean of the second marginal statistic.

'VX' : variance of the first marginal statistic.

'VY' : variance of the second marginal statistic.

'NXY' : creates the matrix of `Nij*Xi*Yj`, which is very useful for calculating covariance.

'CV' : calculating the covariance of a pair `(X,Y)`.

'COR' : linear correlation coefficient of a pair `(X,Y)`.

'LX→Y' : equation of the lines of regression of X in terms of Y.

'LY→X' : equation of the lines of regression of Y in terms of X.

'KX↑A' : fitting a power function `Y=k*X^a` to a statistic.

'KA↑X' : fitting an exponential function `Y=k*a^X` to a statistic.

'SUMT' : sum of the coefficients of a vector or matrix.

'PRODT' : term-by-term product of the coefficients of two vectors (useful for finding variance or covariance).

'MODT' : modifying the terms in a table (matrix or vector) using a certain formula. 'MODT' is useful when fitting power or exponential functions to statistics.

Programs 'SUMT', 'PRODT' and 'MODT' are designed to enable the user to perform "step-by-step" calculations. They may prove useful in directories other than 'STAT2'.

ABSOLUTE FREQUENCIES OF MARGINAL STATISTICS

'FRQX' and 'FRQY' let you find the respective absolute frequencies of marginal statistics X and Y. The result is obtained by addition:

row by row to calculate the absolute frequencies of X,
column by column to calculate the absolute frequencies of Y,
(see the purpose of the matrix 'ΣFRQ' in the introduction).

The result is obtained in vector form at level 1:

```
1: [          ]   'FRQX' or 'FRQY'   1: [  vector  ]
                  ---------------->
```

'FRQX': (Checksum: # 48800d, Size: 46 bytes)

```
« ΣFRQ DUP SIZE 2 2 SUB 1 CON * »
```

'FRQY': (Checksum: # 52973d, Size: 48.5 bytes)

```
« ΣFRQ TRN DUP SIZE 2 2 SUB 1 CON * »
```

Example:

If we take the statistic:

X\Y	10	20	30	40
1	1	7	11	12
5	3	5	0	1
10	0	4	8	8

and put the following matrix in 'ΣFRQ':

```
[[ 1  7  11  12 ]
 [ 3  5   0   1 ]
 [ 0  4   8   8 ]],
```

the vector `[ 1 5 10 ]` in 'X',
the vector `[ 10 20 30 40 ]` in 'Y',

we obtain (in one second)

```
1: [          ]    'FRQX'    1: [ [ 31  9  20 ] ]
                 -------->
1: [          ]    'FRQY'    1: [ [ 4  16  19  21 ] ]
                 -------->
```

ARITHMETIC MEANS OF MARGINAL STATISTICS

'MX' and 'MY' let you find the respective arithmetic means of the marginal statistics X and Y (see purpose of 'ΣFRQ' in the introduction).
The functional diagram is as follows:

```
         ┌─────────────┐   'MX' or 'MY'       ┌─────────────┐
     1:  │             │   ───────────>   1:  │    mean     │
         └─────────────┘                      └─────────────┘
```

N.B: program 'MX' calls 'FRQX' and 'SUMT'. Likewise, 'MY' calls 'FRQY' and 'SUMT'.

'MX':(Checksum: # 19074d, Size: 43.5 bytes)

```
« FRQX DUP SUMT / X DOT »
```

'MY':(Checksum: # 218d, Size: 43.5 bytes)

```
« FRQY DUP SUMT / Y DOT »
```

Example:

If we take the statistic:

X\Y	10	20	30	40
	1	7	11	12
5	3	5	0	1
10	0	4	8	8

we put the following matrix in 'ΣFRQ':

```
[[ 1  7  11  12 ]
 [ 3  5   0   1 ]
 [ 0  4   8   8 ]],
```

and the vector `[ 1 5 10 ]` in 'X',
and the vector `[ 10 20 30 40 ]` in 'Y'.

We obtain (in one second):

```
         ┌─────────────┐      'MX'            ┌─────────────┐
     1:  │             │   ───────>       1:  │     4.6     │
         └─────────────┘                      └─────────────┘
         ┌─────────────┐      'MY'            ┌─────────────┐
     1:  │             │   ───────>       1:  │    29.5     │
         └─────────────┘                      └─────────────┘
```

The arithmetic means of X and Y are therefore X = 4.6 and Y = 29.5.

Note:when dealing with a variable in two statistics (X,Y) consisting of a set of points with an absolute frequency of 1:

X	X1	X2	X3	.....	Xn
Y	Y1	Y2	Y3	.....	Yn

we can still use 'MX' and 'MY' provided that we put the nth order identity matrix in 'ΣFRQ'. It is, however, simpler to apply program 'SUMT' to X and Y before dividing by n.

VARIANCES OF MARGINAL STATISTICS

'VX' and 'VY' let you find the respective variances of the marginal statistics X and Y (see purpose of 'ΣFRQ' in the introduction).

The functional diagram is as follows:

```
                          'VX' or 'VY'
1: [            ]        ------------>     1: [  variance  ]
```

N.B: program 'VX' calls 'FRQX', 'SUMT', 'PRODT' and 'MX'. Likewise, 'VY' calls 'FRQY', 'SUMT', 'PRODT' and 'MY'.

'VX':(Checksum: # 49353d, Size: 65 bytes)

```
« FRQX DUP SUMT / X DUP PRODT DOT MX SQ - »
```

'VY':(Checksum: # 45844d, Size: 65 bytes)

```
« FRQY DUP SUMT / Y DUP PRODT DOT MY SQ - »
```

Example:

If we take the statistic:

X\Y	10	20	30	40
1	1	7	11	12
5	3	5	0	1
10	0	4	8	8

we put the following matrix in 'ΣFRQ':

```
[[ 1  7  11  12 ]
 [ 3  5   0   1 ]
 [ 0  4   8   8 ]]
```

and the vector `[ 1 5 10 ]` in 'X',
and the vector `[ 10 20 30 40 ]` in 'Y'.

We obtain (in under two seconds):

```
                    'VX'
1: [            ]  ------>   1: [        16.44 ]

                    'VY'
1: [            ]  ------>   1: [  88.083333334 ]
```

The variances of X and Y are therefore var(X) = 16.44 and var(Y) ≈ 88.08; we can thus find their standard deviations, which are: $\sigma(X) = \sqrt{16.44} \approx 4.05$ and $\sigma(Y) \approx 9.39$.

Note: when dealing with a variable in two statistics (X,Y) consisting of a set of points with an absolute frequency of 1:
we can still use 'MX' and 'MY' provided that we put the nth order identity matrix in 'ΣFRQ'. It is, however, simpler to use programs 'MODT' (to calculate the vectors of Xi^2 and Yi^2) and 'SUMT', based on Huygens' formula:

X	X1	X2	X3	.....	Xn
Y	Y1	Y2	Y3	.....	Yn

$$\mathrm{var}(X) = X^2 - (X)^2$$ (X = arithmetic mean of X).

CREATING A MATRIX WITH GENERAL TERM Nij*Xi*Yj

'NXY' creates a matrix with the same format as 'ΣFRQ' (see introduction for conventions used) and a general term `NijXiYj`.
The result is put in a variable called 'ΣNXY'.

'NXY':(Checksum: # 44590d, Size: 181.5 bytes)

The stack is not affected by program 'NXY'.

```
«   ΣFRQ    SIZE
    →   d
    «   1    d    1    GET    FOR    i
             1    d    2    GET    FOR    j
                  'ΣFRQ(i,j)*X(i)*Y(j)'    EVAL
             NEXT
        NEXT
        d    →ARRY    'ΣNXY'    STO
    »
»
```

Example:

If we take the statistic:

X\Y	10	20	30	40
1	1	7	11	12
5	3	5	0	1
10	0	4	8	8

we put the following matrix in 'ΣFRQ':

```
[[ 1  7  11  12 ]
 [ 3  5   0   1 ]
 [ 0  4   8   8 ]]
```

and the vector `[ 1 5 10 ]` in 'X',
and the vector `[ 10 20 30 40 ]` in 'Y'.

Program 'NXY' then puts the matrix:

```
[[  10  140  330   480 ]
 [ 150  500    0   200 ]
 [   0  800  2400 3200]]
```

in 'ΣNXY', within 3 to 4 seconds. We then obtain, for example:
as `X(2)=5`, `Y(1)=10` and `'ΣFRQ'(2,1)'=3/`

Note:when dealing with a variable in two statistics (X,Y) consisting of a set of points with an absolute frequency of 1:

X	X1	X2	X3	.....	Xn
Y	Y1	Y2	Y3	.....	Yn

we can still use 'NXY' provided that we put the nth order identity matrix in 'ΣFRQ'. This is not really useful, however, as program 'PRODT' applied to the vectors 'X' and 'Y' is perfectly suited here (as it gives the vector of XiYi).

COVARIANCE AND LINEAR CORRELATION COEFFICIENT

'CV' calculates the covariance `cov(X,Y)` of a variable in two statistics `(X,Y)`.
'COR' calculates their linear correlation coefficient ro(X,Y).
Note that:

$$\mathrm{cov}(X,Y) = \frac{1}{N} \Sigma\ Nij\ (Xi-\overline{X})(Yj-\overline{Y}) = \frac{1}{N} \Sigma\ NijXiYj - \overline{X}\ \overline{Y}$$

and

$$\mathrm{ro}(X,Y) = \frac{\mathrm{cov}(X,Y)}{\sigma(X)\ \sigma(Y)}$$

where X and Y denote the arithmetic means of X and Y and σ(X) and σ(Y) denote their standard deviations.

The functional diagrams are as follows:

```
1: [            ]  --'CV'-->   1: [ cov(X,Y) ]
1: [            ]  --'COR'-->  1: [ ro(X,Y)  ]
```

N.B: 'CV' calls programs 'FRQX', 'MX', 'MY' and 'SUMT'.
'COR' calls programs 'CV', 'VY' and 'VX'.

'CV':(Checksum: # 45578d, Size: 71.5 bytes)

```
«  X  ΣFRQ  Y  *  DOT  FRQX  SUMT  /  MX  MY  *  -  »
```

'COR':(Checksum: # 20309d, Size: 44 bytes)

```
«  CV  VY  √  /  VX  √  /  »
```

Example:

If we take the statistic:

X\Y	10	20	30	40
1	1	7	11	12
5	3	5	0	1
10	0	4	8	8

we put the following matrix in 'ΣFRQ':

```
[ [ 1  7  11  12 ]
  [ 3  5   0   1 ]
  [ 0  4   8   8 ]],
```

and the vector [1 5 10] in 'X',
and the vector [10 20 30 40] in 'Y'.

We obtain, in under two seconds:

```
1: [            ]  --'CV'-->   1: [ 1.133333333 ]
```

and, within four seconds:

```
1: [            ]  --'COR'-->  1: [ 2.97824132168E-2 ]
```

(X and Y are therefore weakly correlated here).

LINES OF REGRESSION OF Y IN TERMS OF X AND OF X IN TERMS OF Y

'LY→X' and 'LX→Y' calculate the equations of the lines of regression of Y in terms of X and of X in terms of Y respectively (also called least squares lines) of the variable in two statistics (X,Y).
The functional diagram is as follows:

```
      ┌──────────────┐   'LY→X' or 'LX→Y'        ┌──────────────┐
1:    │              │   ---------------->   1:  │   equation   │
      └──────────────┘                           └──────────────┘
```

where "equation" is the equation we want to find, in the following form:
    `'y=a*x+b'` for program `'LY→X'`,
    `'x=a*y+b'` for program `'LX→Y'`.
We should also note that the equation of the least squares line of Y in terms of X is:

$$y = \frac{\mathrm{cov}(X,Y)}{\mathrm{var}(X)}(x - \overline{X}) + \overline{Y}$$

and that of the least squares line of X in terms of Y is:

$$x = \frac{\mathrm{cov}(X,Y)}{\mathrm{var}(Y)}(y - \overline{Y}) + \overline{X}$$

(using normal notation).
<u>N.B</u>: 'LY→X' calls 'CV', 'VX', 'MY' and 'MX'.
    'LX→Y' calls 'CV', 'VY', 'MY' and 'MX'.

'LY→X':(Checksum: # 56991d, Size: 82 bytes)

```
«   'y'  CV   VX   /     DUP   'x'   *
    MY   MX   4    ROLL  *     -     +    =
»
```

'LX→Y':(Checksum: # 33808d, Size: 82 bytes)

```
«   'x'  CV   VY   /     DUP   'y'   *
    MX   MY   4    ROLL  *     -     +    =
»
```

<u>Example</u>:
If we take the statistic:

X\Y	10	20	30	40
1	1	7	11	12
5	3	5	0	1
10	0	4	8	8

we put the following matrix in 'ΣFRQ':

```
[ [ 1  7  11  12 ]
  [ 3  5   0   1 ]
  [ 0  4   8   8 ]],
```

`[ 1 5 10 ]` in 'X' and `[ 10 20 30 40 ]` in 'Y'.
We obtain, within four seconds:

```
      ┌──────────────┐       'LY→X'           ┌──────────────────────┐
1:    │              │   ------------>   1:   │ 'y=0.069*x+29.183'   │
      └──────────────┘                        └──────────────────────┘
```

and:

```
      ┌──────────────┐       'LX→Y'           ┌──────────────────────┐
1:    │              │   ------------>   1:   │ 'x=0.013*y+4.220'    │
      └──────────────┘                        └──────────────────────┘
```

FITTING A FUNCTION Y=k*a^X or Y=k*X^a TO A STATISTIC

'KA↑X' and 'KX↑A' can be used to fit:

an exponential function `y=k*a^x` (program 'KA↑X')

a power function `y=k*x^a` (program 'KX↑A'),

to a variable in two statistics `(X,Y)`.

The functional diagram is as follows:

```
1: [          ]   'KA↑X'or 'KX↑A'   1: [ equation ]
                  --------------->
```

where "equation" is the equation of the curve we want to find, in the following form:

`y=k*a^x` for program 'KA↑X'

`y=k*x^a` for program 'KX↑A'.

N.B: 'KA↑X' and 'KX↑A' call 'MODT', 'CV', 'VX', 'MX' and 'MY'.

Important note:

The contents of X and Y are modified while 'KX↑A' is running (only Y is modified in 'KA↑X'). The initial values of the variables X and Y are restored at the end of the program. We therefore have to be careful not to quit the program by pressing "ON" before it has actually terminated. If not, the initial values of X and Y are put in the stack.

'KA↑X':(Checksum: # 35728d, Size: 159 bytes)

```
«  Y     DUP    « LN »   MODT   'Y'   STO   CV   VX   /   EXP
   →     a
   «     'y'    a    MX    NEG    ^    MY    EXP    *    a    'x'    ^    *    =
         SWAP   'Y'   STO
   »
»
```

'KX↑A':(Checksum: # 14153d, Size: 207.5 bytes)

```
«  X    Y     DUP2
   « LN »     MODT    'Y'    STO    « LN »    MODT    'X'    STO
   CV   VX    /
   →    a
   «    'y'   MY    a    MX    *    -    EXP    'x'    a    ^    *    =
        3     ROLLD    'Y'    STO    'X'    STO
   »
»
```

Example: using the data in the example illustrating 'LX→Y' and 'LY→X', we find (within 4 seconds, in 3 FIX mode):

```
1: [          ]     'KA↑X'      1: [ 'y=27.239*1.003^x' ]
                  --------->
and:
1: [          ]     'KX↑A'      1: [ 'y=28.360*x^(-0.024)' ]
                  --------->
```

SUM OF COEFFICIENTS OF A TABLE

'SUMT' calculates the sum Σ of terms in a table A (vector or matrix) as shown in the functional diagram below:

```
1: |  A  |   'SUMT'  ---->   1: |  Σ  |
```

'SUMT':(Checksum: # 7618d, Size: 51 bytes)

```
« OBJ→  OBJ→  IF  2  ==  THEN  *  END
  2  SWAP  START  +  NEXT
»
```

Example: (in under one second)

```
   [[21 48 13 17 44 ]
    [ 8 64 47 11 29 ]     'SUMT'
1:  [13 24  9 55 81 ]]   ------->1:   484
```

TERM-BY-TERM PRODUCT OF TWO VECTORS

'PRODT' calculates the term-by-term product of two vectors `A=[X1,...,Xn]` and `B=[Y1,...,Yn]`. The result is the vector `C=[X1*Y1, X2*Y2, ..., Xn*Yn]`.

The functional diagram is as follows:

```
2: |  A  |   'PRODT'   2: |     |
1: |  B  |   ------>   1: |  C  |
```

'PRODT':(Checksum: # 27673d, Size: 122 bytes)

```
« DUP  SIZE  1  GET
  →  a  b  n
  «  1  n  FOR  i  'a(i)*b(i)'  EVAL  NEXT
     n  →ARRY
  »
»
```

Example: (in one second)

```
2: [ 12  7  31  9 4 ]   'PRODT'  2:
1: [  4 11   2  1 8 ]   ------>  1: [ 48 77 62 9 32 ]
```

APPLYING A SINGLE PROCEDURE TO THE COEFFICIENTS OF A TABLE

'MODT' enables you to modify the coefficients of a table A by applying a single transformation. If B is the table obtained, we get the following functional diagram:

```
2: |     A      |   'MODT'    2: |        |
                 ----------->
1: | procedure  |             1: |   B    |
```

where "procedure" is a program designed to be applied to each of the elements in A. This program is supposed to apply to the element at level 1 of the stack and thus to give a numerical result at the same level.

'MODT':(Checksum: # 48215d, Size: 110 bytes)

```
«  →  p
   «  OBJ→   DUP   →   d
      «  OBJ→   IF   2   ==   THEN   *   END
         →  n
         «  1   n   START   p   EVAL   n   ROLLD   NEXT
            d   →ARRY
         »
      »
   »
»
```

Example 1: (in one second)

```
                                           (Result in 3 FIX mode)
2: | [ 1.7   3.5   4.25 ] |   'MODT'    2: |                       |
                           ----------->
1: |        « LN »        |             1: | [0.531 1.253 1.447]   |
```

Example 2: (in one second)

```
   | [[ 15   21   33 ]   |
   |    12   36   99 ]   |
2: |     6   45   12 ]]  |   'MODT'        | [[  25   49    121 ]  |
                          ----------->     |  [  16  144   1089 ]  |
1: |    « 3 / SQ »       |             1:  |  [   4  225     16 ]] |
```

(all coefficients were divided by 3 then squared).

Note:'MODT' is useful when attempting to find a relation of the type `Y=K*X^α` or `Y=K*A^X` between two statistics `X` and `Y`. The program enables us to transform the vector of `Xi` and/or `Yi` into that of `LN(Xi)` (or `LN(Yi)`) and to find the linear correlation between these two new variables (programs 'CV', 'COR', 'LY→X', etc).

DATABASES

The HP48's large memory capacity means that it is able to manage databases. You will not, of course, be able to enter huge amounts of data, but for a pocket calculator you can obtain very impressive results.

A database is a set of records (that are usually in some way inter-related) in which we can add, edit, delete, read, sort and select records.

A database will be stored in the HP48's memory as a list in the following form:

{ C1 C2 C3 ... Cn },

where n is an integer representing the number of elements in the list and C1, C2, ..., Cn are the various records.

A record is a string of characters.

For such a string to be displayed in full on screen, it should not exceed 7 lines in length, each line containing no more than 22 characters (line feeds can be inserted with the NEWLINE command).
However, as we shall see, longer strings can be used with the VISIT option of program 'READ'.

A database must be put in a variable and may be consulted using the name of the variable.

Here is the list of the programs in the 'DATA' directory:

'ADD' : adds a record.

'READ' : reads the database and edits or deletes records.

'SORT' : sorts the database.

'FIND' : finds the first record containing a given sub-string (and reads the database starting from this record).

'SELEC' : selects records containing a given sub-string, then reads the database created from these records.

ADDING A RECORD

'ADD' lets you add a record to an existing database or create a new database by entering a new record.
This gives us two possible functional diagrams:
NAME denotes the name of the database to be updated or created.

Updating:

```
2: |  NAME  |    'ADD '      2: |        |
1: |  REC   |  ---------->   1: |  NAME  |
```

Here REC is the string of characters representing the new record.

Creating:

```
                 'ADD'
1: |  NAME  |  ---------->   1: |  NAME  |
```

Here a "New Object" message is displayed to allow you to enter the new record.
The keyboard switches to Alpha mode: when you confirm the object to be entered by pressing ENTER, it is converted into a string of characters (you do not need to include inverted commas).

If the database is properly sorted, the new record is entered in such a way that the database is resorted.

'ADD': (Checksum: # 22744d, Size: 227 bytes)

```
«  IF    DUP    TYPE    2    ≠    THEN
         "New Object:"    { "" α }    INPUT
   END
   →   c
   «   IFERR   DUP   RCL   THEN
             DROP   c   1   →LIST
       ELSE
             ""   SWAP   OBJ→   →   n
             «   c   1
                 WHILE
                     DUP   n   ≤   OVER   3   +   PICK   c   >   AND
                 REPEAT   1   +
                 END
                 ROLLD   n   1   +   →LIST   SWAP   DROP
             »
       END   OVER   STO
»  »
```

READING AND EDITING THE DATABASE

'READ' allows you to consult and edit the contents of a database. The functional diagram is as follows:

```
                                            2: |  NOM  |
                          'READ'
1: |  NOM  |          ------------>         1: |  Num  |
```

where NAME is the name of the database and Num is the number of the record that has been located (you can then put the record in the stack with **GET**).

N.B: You can also call 'READ' by putting the name of the database at level 2 and the record number at level 1. The program then reads the database from the record whose number you have entered (this can be useful when working with large databases if you want to locate a record that is a long way from the start).

Program 'READ' halts as soon as you call it up and the contents of the record are displayed on screen with the following menu:

->	<-	->->	VISIT	DEL	OK

1) Press "→" to display the next record. If you are on the last record, this will take you back to the start.

2) Press "←" to display the preceding record. If you are on the first record, this takes you to the end.

3) Press "→→" to scroll through the database. When you reach the end, scrolling continues from the start. Press any key (except ON) to halt scrolling.

4) Press "VISIT" to display the record on the command line. You can thus consult the record (especially if it is too large to be fully displayed on screen).
You can also edit the record at the same time.
Press ENTER to terminate consultation/editing.
If you have edited the record, the program asks you to confirm by pressing Y (yes) or N (no).
By confirming, you replace the old record with the newly edited one.
If not, any changes made are cancelled.
In both cases, you can then continue reading the database.

5) Press "DELET" to delete the record you are consulting. To avoid deleting by mistake, the program asks for confirmation by Y (yes) or N (no). In both cases, you can then continue reading the database.

6) Press "OK" to finish reading. The name of the database is then at level 2 and the record N° at level 1. Press 'READ' again to continue reading from the point where you left the database.

Notes:

If you delete the last record of a database, the database is purged from the directory.
When scrolling, 0.2 seconds elapses between screens (see .2 WAIT instruction, which can be modified, below).

'READ': (Checksum: # 48934d, Size: 651 bytes)

```
«   IF    DUP    TYPE   6   ==   THEN   1   END   OVER   RCL   SIZE
    →    i    n
    «    «    i    +    n    MOD    1    +    'i'    STO    DUP
              i    GET    CLLCD    1    DISP    3    FREEZE
         »
         «    "Confirm? (Y/N) :"    { "" α }
              INPUT    "Y"    SAME
         »    →    ii    ok
         «    -1    ii    EVAL
              {    {    "→"  «    0    ii    EVAL    »    }
                   {    "←"  «    -2   ii    EVAL    »    }
                   {    "→→"
                        «    DO    0    ii    EVAL    .2    WAIT
                             UNTIL    KEY    END    DROP
                        »    }
                   {    "VISIT"
                        «    DUP    i    DUP2    GET    ""    OVER
                             { -1 }    +    INPUT
                             IF    DUP    ROT    SAME    THEN    0
                                   ELSE    ok    EVAL
                             END
                             IF    THEN    PUT    ELSE    3    DROPN    END
                             -1    ii    EVAL
                        »    }
                   {    "DEL"
                        «    IF    ok    EVAL    THEN
                                   IF    n    1    >    THEN
                                         DUP    RCL    DUP    1    i    1    -    SUB
                                         SWAP    i    1    +    OVER    SIZE    SUB
                                         +    OVER    STO    'n'    1    STO-
                                   ELSE    PURGE    2    MENU    KILL
                                   END
                             END    -1    ii    EVAL
                        »    }
                   {    "OK"    CONT    }
              }    TMENU    HALT    2    MENU    i
»   »    »
```

SORTING THE DATABASE

'SORT' is used to sort a database in ascending alphabetical order (in ascending order of ASCII codes to be more precise).

Sorting is only necessary if certain records have been edited (VISIT option in program 'READ'), thus cancelling the order established automatically when using 'ADD'.

The functional diagram is simple:

```
    ┌──────────────┐     'SORT'        ┌──────────────┐
1:  │     NOM      │  ──────────>  1:  │     NOM      │
    └──────────────┘                   └──────────────┘
```

where NAME represents the name of the database.

'SORT': (Checksum: # 28921d, Size: 142.5 bytes)

```
«  DUP   RCL   OBJ→
   →  n
   «  IF  n  1  >
      THEN
         2  n  FOR      i
            i  ROLL  i  1  +
            WHILE  DUP  PICK  3  PICK  <
            REPEAT  1  -  END
            1  -  ROLLD
         NEXT
      END
      n  →LIST  OVER  STO
   »
»
```

<u>Note</u>: It is impossible to estimate the time it will take to sort a database. This will depend both on the size of the database and on how sorted or unsorted it already is. A database containing 50 records should be able to be sorted within about 20 seconds.

FINDING A RECORD

'FIND' locates the first record containing a given sub-string in a database by searching sequentially.

There are two possible functional diagrams:
NAME denotes the name of the database.

First case:

```
2: | NOM |
1: | CHN | .
```

Here CHN is the string of characters to be located.

Second case:

```
1: | NOM |
```

In this case the message "search first:" asks you to indicate the object to be located (which is then automatically converted into a string of characters). The keyboard switches to Alpha mode and you enter the object to be located (without including inverted commas) and then confirm by pressing ENTER.

The message "In progress..." is displayed as the program searches.

If the sub-string is found in a record, you can then start reading the database from that record.

If not, a "No occurences" message is displayed.

'FIND': (Checksum: # 38062d, Size: 230.5 bytes)

```
«  IF    DUP    TYPE    2    ≠    THEN
         "Search first:"  {  ""  α  }    INPUT
   END
   OVER    RCL    SIZE
   →    ch    n
   «    CLLCD    "In progress..."    1    DISP
        0
        DO    1    +
        UNTIL
             DUP2    n    MIN    GET    ch    POS    OVER    n    >    OR
        END
        IF    DUP    n    >    THEN
             DROP    "No occurrences"    DOERR
        ELSE
             READ
        END
   »
»
```

SELECTING A RECORD

'SELEC' locates <u>all</u> the records containing a given sub-string in a database by searching sequentially.

There are two possible functional diagrams:
<u>NAME</u> denotes the name of the database.

<u>First case</u>:

```
   +-----------------+
2: |       NOM       |
   +-----------------+
1: |       CHN       |
   +-----------------+
```

Here <u>CHN</u> is the <u>string of characters</u> to be located.

<u>Second case</u>:

```
   +-----------------+
1: |       NOM       |
   +-----------------+
```

In this case the message "search:" asks you to indicate the object to be located (which is then automatically converted into a string of characters). The keyboard switches to Alpha mode and you enter the object to be located (without including inverted commas) and then confirm by pressing ENTER.

The message "In progress..." is displayed as the program searches.

If the sub-string is found in a record, then 'SELEC' creates a new database called '°' containing all the records in which the sub-string is located.
You can then read the database '°'.
Once you have finished reading, the variable '°' is not purged.

If no sub-string is found, a "No occurences found" message is displayed.

'SELEC': (Checksum: # 39471d, Size: 257 bytes)

```
«   IF     DUP    TYPE    2    ≠     THEN
           "Search :"      { "" α  }     INPUT
    END
    OVER    RCL    SIZE    →    ch    n
    «     CLLCD    "In progress..."    1    DISP    {}
          1    n    FOR    j
               OVER    j    GET
               IF    DUP    ch    POS    THEN    +
                     ELSE   DROP
               END
          NEXT
          IF    DUP    SIZE    THEN
                '°'    STO    '°'    READ
          ELSE    DROP    "No occurrences"    DOERR
          END
»   »
```

INDEX OF PROGRAM NAMES

INDEX OF PROGRAM NAMES (Cont.)

INDEX OF PROGRAM NAMES (Cont.)

INDEX OF PROGRAM NAMES (Cont.)

VOLUME 1 - CONTENTS

NOTES

NOTES

NOTES

NOTES

Achevé d'imprimer par

PARA
GRAPHIC

31240 L'UNION (Toulouse)
Tél. (16) 61.74.27.67
Dépôt légal : Juillet 1991

TABLE OF CONTENTS

ISBN 2-908791-04-8